TAMPERE UNIVERSITY OF TECHNOLOGY

RUPESH DEV

UI TEST AUTOMATION IN SYMBIAN CAMERA SOFTWARE DEVELOPMENT

Master of Science Thesis

Examiners: Professor Jarmo Harju
Adjunct Professor Mika Katara
Examiners and topic approved in the Faculty of Computing and Electrical Engineering Council meeting on 04.05.2011

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
DEV, RUPESH: UI Test Automation in Symbian Camera Software Development
Master of Science Thesis, 52 pages, 2 Appendix pages
June 2011
Major: Communications Engineering
Examiners: Professor Jarmo Harju, Adjunct Professor Mika Katara
Keywords: GUI Automation, Model based testing, Symbian devices

Software testing is one of the most cost-intensive tasks in the modern software production process. Software testing needs to be effective not only at finding the defects, but also in performing the tests as quickly and cheaply as possible. Automation in software testing has been used widely to achieve faster test results in limited time and effort.

This thesis tries to demonstrate model based testing (MBT) approach as one of the most promising automation methods developed in recent times. Model based testing is a relatively new software test automation methodology that automates not only the test execution, but also the test generation. The basic idea is to create formal test models which possess the logic of the system to be tested and generate tests based on the models.

This thesis also presents an implementation of a model based approach in automating the software tests. Scope of the thesis is to carry out only UI related test automation. The target system to conduct the test runs is Symbian OS. In the case studies section, the entire procedure of automating the test cases has been explained. Only camera and messaging related test cases have been automated so far. The end devices selected for executing the test runs are Nokia smartphones, namely N8 and E7.

This thesis also analyzes potential problems in deploying model based approach in wider scale and at the same time also proposes an intermediate solution for deploying it in industries within small teams. At the end, the thesis concludes by recommending ways to implement MBT approach in other mobile software platform like Windows Phone.

ACKNOWLEDGEMENTS

This thesis work has been accomplished with joint cooperation of Nokia Corporation and TEMA toolset development group in Tampere University of Technology. I would like to express my sincere thanks to all the people who contributed their valuable time and guidance to make this thesis work implementable.

First and foremost I owe my deepest gratitude to my supervisor Mika Katara, whose supervision and support from the preliminary to the concluding level enabled me to develop an understanding of the subject. I also express my heartfelt thanks to my manager Petri Soininen from Nokia Corporation for his encouragement and assistance on providing resources for the thesis. I am also thankful to Professor Jarmo Harju for counseling and advising me during thesis writing.

My special thanks to Antti Jääskeläinen, Tommi Takala, and Henri Heiskanen for helping me to sort out complexities related to TEMA toolset. From Nokia side, I am equally thankful to Petri Kiiskinen whose help and guidance with Testability driver tool is appreciable. Also vote of thanks to Ville Ilvonen for making it possible to reach out Testability driver team. I am equally grateful to all of my teammates in Nokia, mainly Sachin Nayak for providing me all kinds of moral support and advice throughout the thesis completion. I am equally indebted to Pekka Kauppila for his support and guidance during initial phase of the thesis. I am indebted to many of my colleagues for supporting me in numerous ways. Especially Kati Pyhälä, Tomi Pihlainen, and Kari Jussila for inspiring me and appreciating the thesis work. I am also thankful to Euan Barron for his suggestions during thesis finalizing.

Finally I want to thank my parents; Mother, Brother, and late Father for their blessings and love to stand with me against each obstacle. I am grateful to God, the creator and the guardian to whom I owe my very existence. Also, I am thankful to my girl friend for her understanding and moral support during thesis writing.

Rupesh Kumar Dev
10th May 2011

CONTENTS

ABBREVIATIONS AND DEFINITIONS

Action machine	A model component that describes the functionality of the SUT at the level of action words.
Action word	A high-level action executable by the SUT, implemented with keywords.
BBT	Black Box Testing.
Coverage language	The syntax of forming coverage requirements.
Coverage requirement	A formal test objective that defines the ending criteria of a test run and acts also as a guideline to the guidance algorithm with respect to the actions to be executed in order to fulfill the ending criteria.
Data table	A data structure containing the external data to use in data statements in TEMA models.
GUI	Graphical User Interface.
Initialization machine	A model component that defines necessary initialization procedures for the SUTs.
Localization data	GUI texts in some specific language.
LPT	Long Period Testing.
MBT	Model Based Testing.
MeeGo	Linux-based open source mobile operating system project.
OS	Operating System.
Refinement machine	A model component that contains keywords implementing action words.
SMS	Short Message Service.

SUT	System under Test.
SW	Software.
Symbian	An Operating system used for mobile phones, owned by Nokia Corporation.
Symbian S^3	Latest Symbian operating system version, officially released in Q4 2010.
TD	Testability Driver, an open source test automation tool owned by Nokia.
TEMA	Test modeling using Action Words, a model based testing tool.
Test model	A formal model that describes the functionality of the SUT in model based testing.
Test modeler	A person who builds a model for test execution.
Ubuntu	An operating system based on the Debian GNU/Linux.
UI	User Interface.
USB	Universal Serial Bus.
Use case	An action sequence that an actor performs within a system to accomplish a particular goal.
WLAN	Wireless Local Area Network

1. Introduction

"*In business, the competition will bite you if you keep running; if you stand still, they will swallow you*. " – William Knudsen [1]

Unlike other fields which are more predictable, technology is moving rapidly, and its developmental pace has been exponential. Whether it is small thumb shape flash disk replacing a huge storage disk drive; or a small cell phone performing thousand times better than a giant handset, technological change has already witnessed several new dimensions in this dynamic era.

Emergence of the Internet and its wide use has united the whole world into a single global village. It has made people more aware of new technologies which have raised demands as well as choices. To meet the demands, satisfy consumers, and not to get lost in a crowd of competitors, one must produce user friendly, reliable, qualitative and low cost products, whether it is hardware, software or a mix of both. For that, new technology, methodology, tools and processes must be adopted which can fulfil the needs and the requirements of users to stay ahead in the race.

Automation in SW testing is one of such methodologies. Automating software testing can significantly reduce the effort required for adequate testing, or significantly increase the testing which can be done in limited time. Tests can be run in minutes that would take hours to run manually. Automated tests are repeatable, using exactly the same inputs in the same sequence, something that cannot be guaranteed with manual testing.

The main purpose of this thesis is to implement model based testing for automating the software testing procedure. For this, a model based test tool TEMA has been used along with a keyword based test tool Testability driver. Both test tools perform together to automate test steps on SUTs. The SUTs used in testing are Symbian smartphones, namely E7 and N8. For test automation purpose, this thesis mainly targets the cases related to *Camera* and *Messaging*. For example: most basic use of camera to capture images, recording videos etc. is automated. In addition this thesis aims to work with two entirely different test automation tools, and demonstrates the way to establish a communication channel between them. This thesis work provides an overall idea of test automation; showing model based testing approach to be one of the most efficient and viable approaches for UI automation in the context of Nokia smartphones.

By the end of this thesis, a reader will have an overall idea of how to make use of model based approach and what new changes need to be done to update and improve the current practice of keyword based automation.

The thesis comprises of six different chapters. Chapter 2 describes different test approaches and their definitions. It includes some specific testing types that would be used as an example for UI automation, for example: long period testing, and parallel testing. It also explains about the model based test design and the corresponding automation tool designed for it. Benefits and challenges of implementing automation with model based testing will be also mentioned in short.

Chapter 3 describes the goals set before commencing the real test run. It describes the automation tool being used, and explains the technical knowhow of these tools. A detailed explanation on these tools is elaborated, along with the possible flaws in each of them. The functional architecture of these tools is also shown together with some screenshots of the GUI interfaces being used to analyze the scripts and results obtained.

Chapter 4 describes methods followed to perform the model based testing. It describes the model's structure and execution as a whole. It also explains about execution logic used, Linux host setup technicalities, SUT setup preconditions, tool setup practice, and finally adapter's role is described in a practical way.

Chapter 5 shows the real implementation on the target environment set, as defined in the chapter 4. Single/SUT test cases as well as multi/SUT test cases have been automated with different test case scenarios. Alternative keyword based script for the same cases have been also kept for comparison. Only the cases related to camera image and video captures will be within target. Thorough analysis of results will be discussed comparing the models and keywords.

The conclusions are in Chapter 6. This chapter includes information on how further development of this thesis project can be achieved and tuned to achieve better performance.

Appendix A is located at the end of the thesis. It shows information on a sample log file generated during the text execution.

2. Software Testing and Test Automation

"*Testing is a process of gathering information by making observations and comparing them to expectations*." (Dale Emery and Elizabeth Hendrickson) [2]

Testing is an inevitable part of the software engineering process. The purpose of software testing is to find faults in the software and to verify that the developed product fulfils the requirements set at the beginning of the software process. Software testing can be both manual and automated. Manual testing could be appropriate to some designated test sets and domains, but it fails behind especially when the same tests need to be executed quite often and for a long period of time. This results in manual testing being more time consuming and an expensive activity.

Software testing accounts for a large percentage of effort in the software development process which requires systematic planning, execution and control to make it more productive. It is a broad area, which involves many other technical and non-technical sectors, such as specification, design and implementation, maintenance, process and management issues in software engineering.

2.1 Software Testing in General

In general, the organizations perform software testing to identify defects in the software. Defects in software testing can be defined as variance from requirement or user expectation. There are several methods in software testing which can be followed to discover the possible defects in software.

Software testing has been categorized into many forms and types depending on the need and variation of test cases. A section below describes various kinds of testing strategies through Figure 2.1. In the figure, one axis shows the scale of the System under Test (SUT), ranging from small units up to the whole system. Another axis shows the different characteristics that we may want to test, including the most common *functional testing*. The third axis shows the kind of information we use to design the tests.

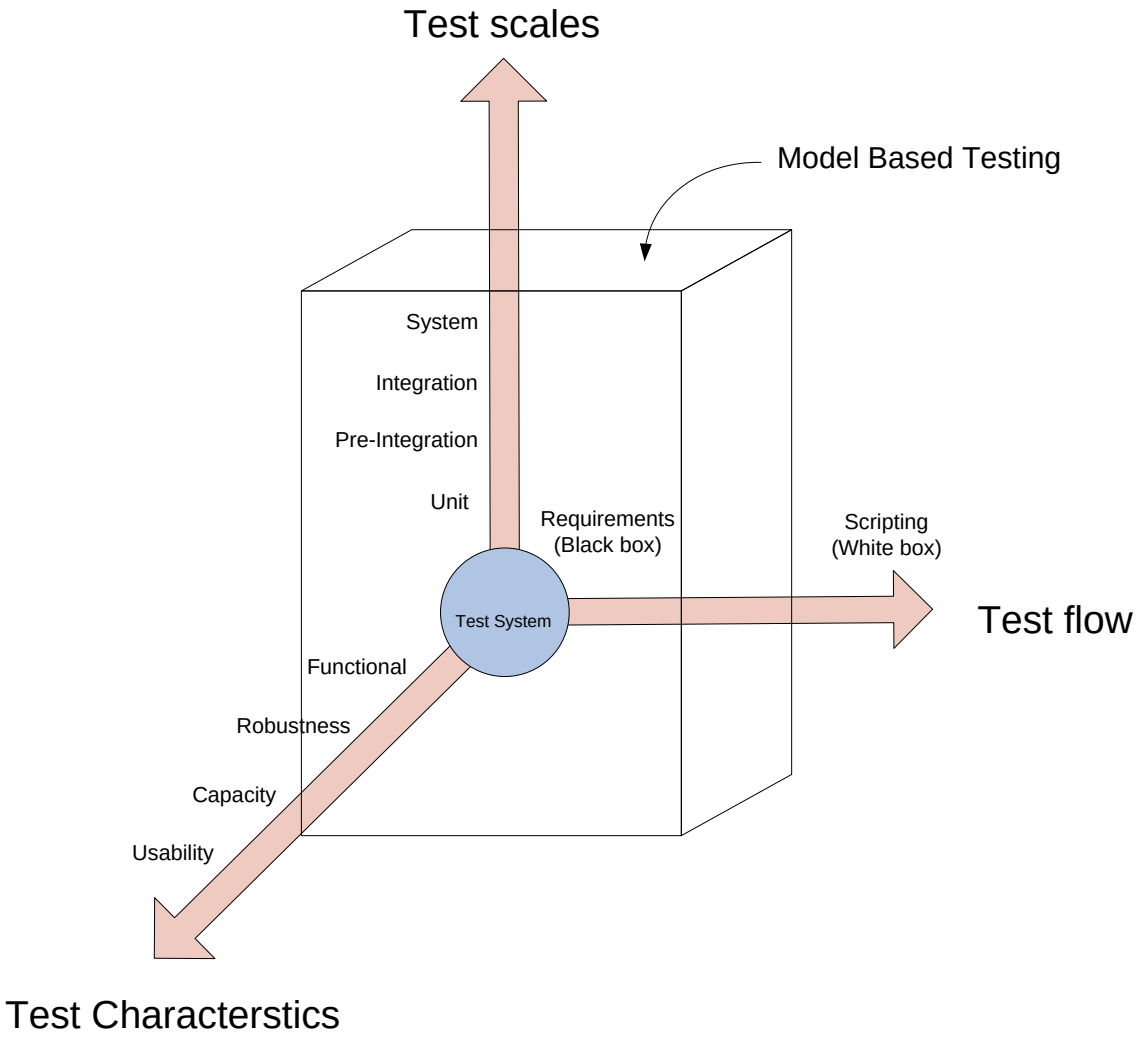

Figure 2.1: Different kinds of Testing, adapted from [3]

Classification based on Scale of the System under Test (SUT)

Unit Testing
White-box testing methodology applies to unit testing in which functionality of code is tested generally at function and/or class level. Developers write the code to test and verify the functionality of a piece of software.

Component Testing
Test method where each component/subsystem is tested separately.

Integration Testing
Integration testing is a testing method in which modules are combined and tested as a group. Modules are typically code modules, individual applications, client and server applications on a network, etc. Integration testing follows unit testing and precedes system testing. [4.]

System Testing
System testing falls within black-box testing and is done to ensure that the entire software system is in compliance with the requirements specification. It does not require any knowledge of inner design (logic and/or code) of the system. [5; 4.]

Classification based on Characteristics to test

Functional Testing
Testing the features and operational behavior of a product to ensure they correspond to its specifications. Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions. [4.]

Robustness Testing
Robustness testing aims at finding errors in the system under invalid conditions, such as unexpected inputs, unavailability of dependent applications, and hardware or network failures. [6, p. 6]

Performance Testing
Performance testing is done to verify and validate systems response, quality and reliability. The system is tested in various scenarios to check its speed and to determine that how much stress or load the system can stand [4]. Power consumption testing is one of the examples, which is one of the important things in the smartphone business.

Usability Testing
Usability testing focuses on finding user interfaces problems, which may make the SW difficult to use or may cause the users to misinterpret the output. [6, p. 6]

Classification based on test design information

Black Box Testing (BBT)
Black Box Testing is a testing strategy based on requirements and specifications. Black box testing requires no knowledge of internal paths, structures, or implementation of the software

under test. This testing methodology looks at what are the available inputs for an application and what the expected outputs are that should result from each input. [7.]

An example of a black box testing process would be a test automation tool used by a tester. A tester uses the test automation tool with the pre-written test scripts and executes them. But, a tester does not necessarily understand any inherent technicalities about the tool and script being used.

White Box Testing

White Box Testing is a testing strategy based on internal paths, code structures, and implementation of the Software under Test. White box testing generally requires detailed programming skills in most of the cases. [7.]

An example of a white box testing process would be the same test automation tool used by a programmer. A programmer has an understanding of the inherent implementation details and also possesses knowledge of test scripting. He/she can visualize the working phenomenon of a test script easily and also update it according to the requirements.

Grey Box Testing

Grey box testing is a software testing technique that uses a combination of black box testing and white box testing. Grey box testing is not a complete BBT, because the tester does know some of the internal workings of the software under test. In grey box testing, the tester applies a limited number of test cases to the internal workings of the software under test. In the remaining part of the grey box testing, one takes a black box approach in applying inputs to the software under test and observing the outputs. [8.]

The following section elaborates three different types of system testing approaches; on which automation works were done extensively while preparing this thesis. This form of testing are carried out in a daily or weekly basis to hunt the potential bugs in the SW itself. Also, hardware related issues sometimes affect the execution of SW testing. Below, these testing methods are mentioned in short, and will be elaborated more in context of real test cases discussed in Chapter 4, and 5.

2.2 Software testing in context of Camera

In general, when a new Camera SW is released, it undergoes many different kinds of testing practices. Some of such prominent testing methods executed for camera SW are mentioned below. These methods are explained on the basis of how it is utilized while testing corresponding camera related tests.

2.2.1 Stress testing

Stress tests force programs to operate under limited resource conditions. The goal is to push the upper functional limits of a program to ensure that it can function correctly and

handle error conditions gracefully. Examples of resources that may be artificially manipulated to create stressful conditions include memory, disk space, and network bandwidth. [9.]

Practical Use Case: "*Capturing many images in different light conditions, without any storage media inside a phone"*.

2.2.2 Parallel Testing

Parallel testing involves testing multiple products or subcomponents simultaneously. The main purpose to conduct the parallel testing is to check the concurrency issues. For example: a testing that involves starting a music player followed by opening a camera application. When two or more applications are opened simultaneously, none of them should get affected. It implies that one of the applications must remain opened in the background.

Also the majority of nonparallel test systems test only one product or subcomponent at a time, leaving expensive test hardware idle more than 50 percent of the test time. Thus, with parallel testing, we can increase the throughput of manufacturing test systems without spending a lot of money to duplicate and fan out additional test systems. [9.]

Practical Use Case: "*Recording videos and capturing images simultaneously in two different Devices Under test (DUT), each having different SW versions"*.

2.2.3 Long Period Testing

Long period testing is sort of a performance testing, where DUTs are automated to run for infinitely long time. Devices are tested for longer period to investigate on the issues like memory leaks, software freezing, and hardware failure. These issues otherwise can never be seen during normal testing period. LPT has become a regular target of automation for every SW company to assess the performance of SW beforehand.

Practical Use Case: *"Capturing many Images and videos in a loop until the memory card/storage media gets full i.e. running test for more than 24 hours"*.

Among three different testing approaches mentioned above, only parallel testing and long period testing will be considered as per the scope of this thesis. The case studies in Chapter 5 of this thesis describe the methods used in carrying out these testing strategies. Basically camera related test cases including image captures and video recordings fall in the category of long period testing, whereas multi-phone messaging is a good example of parallel testing.

Next section describes different methods of test automation used in practice.

2.3 Classic methods of SW Test Automation

This section describes several classic testing processes that are widely used in SW industry. We will start describing manual testing process followed by several testing processes that use automated test execution. A diagram will be used to elaborate each testing process, and notations used in these process diagrams are shown in Figure 2.2.

Some of the notations that are used to define the diagrams are as follows:

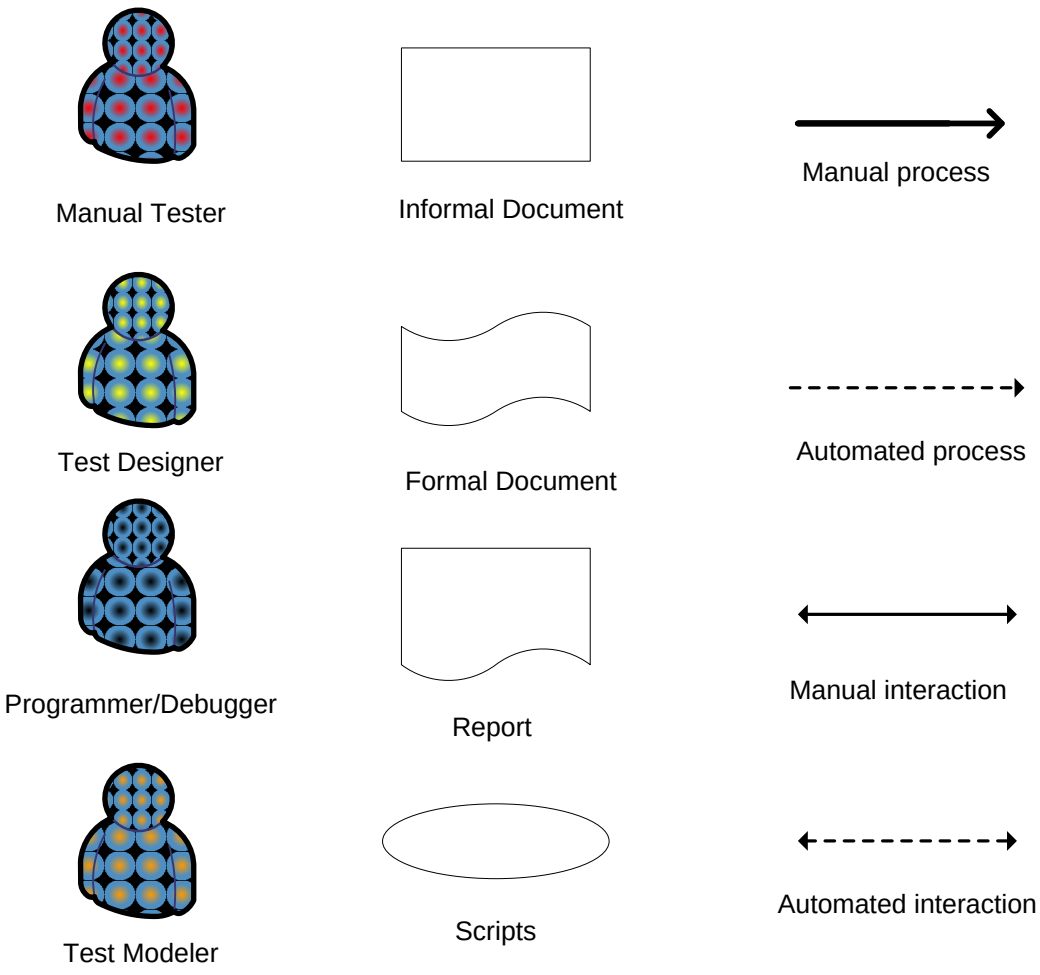

Figure 2.2: Notations used in process diagrams, adapted from [6, p. 20]

Manual Tester: Manual testers perform SW testing activities manually. They put themselves as an end user, and use most of all features of the application to ensure correct behavior. To ensure completeness of testing, the testers often follow a written test plan that leads them through a set of important test cases.

Test Designer: The Test Designer role is responsible for defining the test approach and ensuring its successful implementation. The role involves identifying the appropriate techniques, tools and guidelines to implement the required tests, and to give guidance on the corresponding resources requirements for the test effort. [10.]

Test Modeler: The Test Modeler builds the logic behind the models. A well balanced model in-line with the requirements of the project is needed. He/She possess a skill of creating a model. The models need to be uploaded successfully to automate the test cases later.

Programmer/Debugger: Programmer works on creating a script, execute them, check the results, and if not appropriate updates the script again. He/She also has a deep knowledge on technical knowhow of the tools being used for automation. A Debugger analyzes through the test report generated during test execution. These test reports are basically the logs which record all events being executed.

2.3.1 Manual Testing Process

Manual testing is an earliest style of testing which is still used widely. The test design is done manually based on informal requirements documents. The test plan gives high-level overview of the testing objectives.

The output of the design stage is a human-readable document that describes the desired test cases. The test execution is also done manually as shown in Figure 2.3. For each test case, the manual tester follows the step of that test case, interacts directly with the SUT, compares the SUT output with the expected output, and records the test verdict.

This manual test execution process is repeated each time a new release of the SUT needs to be tested. This can become a boring and time consuming task if performed repeatedly. Since there is no automation of the test execution, the cost of testing each SUT release is constant and large. In fact, the cost of repeating manual test execution is so high that, to keep testing costs within budget, it is often necessary to cut corners by reducing the number of tests that are executed. This can result in SW being delivered with incomplete testing, introducing significant risk regarding product maturity, stability, and robustness.

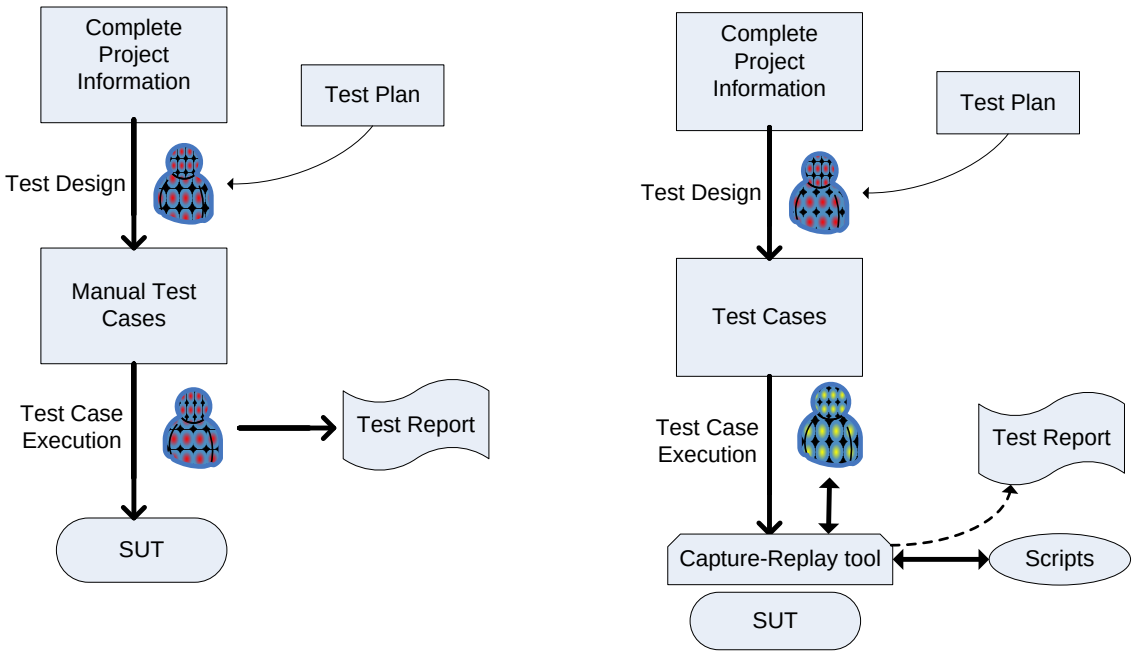

Figure 2.3: Manual testing process (left) and a Capture/Replay testing process (right), adapted from [6, p. 21]

The figure above depicts the differences in the manual and capture/replay testing process. Details of capture/replay testing process are explained in a section below.

2.3.2 Capture/Replay Testing Process

Capture/Replay testing attempts to reduce the cost of the test re-execution by *capturing* the interactions with the SUT during one test execution session and then replaying those interactions during later test execution sessions. But test cases are still designed manually.

Difference to manual testing with this approach is that a manual tester need not necessarily test the repetitive test cases unless the SW interface or any other parameters like UI has changed in SUT. The interaction with a SUT is managed by a tool, namely capture/replay tool. When a new SW release must be tested, this tool can attempt to rerun all the recorded tests and report which ones fail. To rerun each recorded test, the tools send the recorded inputs to the SUT and then compare the new output with the recorded outputs from the original test execution. Figure 2.3 describes Capture/Replay testing process. [6.]

Flaws: Performance of Capture/Replay testing process is very fragile in nature. Change in layout of window can diminish every test cases designed so far.

2.3.3 Script Based automation process

Script based automation uses *test scripts* to automate the test execution in a SUT as shown in Figure 2.4. A test script can contain one or more test cases specification inside

it. In terms of camera based test automation, it can be *launching a camera, capturing an image, switching to video mode, tapping the screen* etc.

The test scripts may be written in some standard programming or scripting language. A scripting language is a set of commands for controlling some specific software applications, hardware or operating system. The script based testing approach solves the test execution problem by automating it. Each time that we want to rerun the tests for regression testing, this can be done for free by just running the test scripts again.

However, this increases the test maintenance problem because the test scripts must evolve not only when some requirements change, but also whenever some implementation details change. (For example: when some parameters change in the API used to stimulate the SUT). In technical terms, we define it as *Lack of Abstraction* in the recorded tests.

2.3.4 Keyword-Driven Automation process

Keyword-Driven Automation targets to overcome the maintenance problems in the script based automation by raising the abstraction level of the test cases.

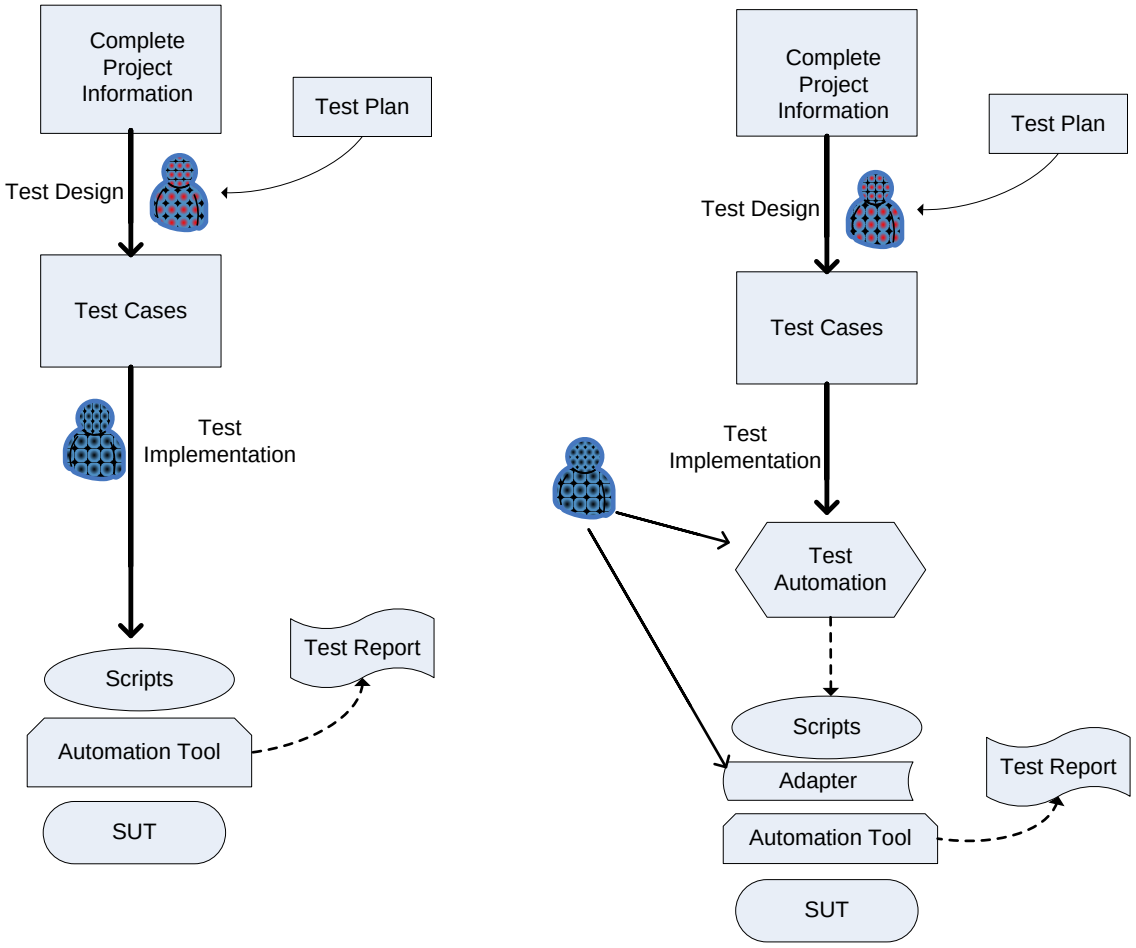

Figure 2.4: Script based (left) and keyword-driven automation process (right), adapted from [6, p. 23]

Keyword driven automation involves using sequence of *action keywords* in the test cases, in addition to data. As shown in Figure 2.4, the code adapter acts as an interface between script and test execution tool. Adapter allows the tool to translate a sequence of keywords and data values into executable tests. One example of keyword-based testing automation is *Testability driver*, which is a tool open sourced by Nokia. It can be used for test automation for Qt applications running on several platforms which has Qt installed.

Testability driver has class library implemented in Ruby [11] language and provides access to communicate with the target SUT in Ruby. Action keywords written in Ruby have less dependency with the type of design or UI interfaces of the SUT. Hence, the same script in Ruby can be used repeatedly for different SUTs or release versions resulting on high level of abstraction of the test cases. This eventually reduces the maintenance problems because the test cases can often be adapted to new version of SUT environment.

Despite of all those higher abstraction, keyword-based automation process still involves manual participation to some extent. For example: Test data are designed manually, as well as verification of test coverage with respect to requirements has to be done and maintained manually.

In the upcoming sections, the possible problems in automation and its solutions will be discussed.

2.4 Model Based Testing

Model based testing is the automatic generation of efficient test procedures/vectors using models of system requirements and specified functionality. [12.]

Unlike previously mentioned automation processes, with *Model Based Testing* both test generation and test execution are automated. The test designer writes an abstract model of the SUT, and then the MBT tool generates a set of tests with that model.

The MBT can be divided into two different categories; online and offline testing. Offline testing signifies test suite generation from the model and its later execution. The export format of generated test cases depends on the used execution tool, and can be, for example a test script. In the online test generation approach, tests are generated and executed in same time. With online testing, it is possible to react to continual changes, and make autonomous decisions. This makes it possible to test non-deterministic systems and run infinite test runs. [13; 14.]

2.4.1 Offline approach

With offline MBT approach, test generation and execution are carried out separately. Offline MBT testing process is described in Figure 2.5. The target system’s behavior is

described in an informal requirements document. A model for test generation is made from the requirement specification. The model is imported to the test generator. The test generator generates test suites from the model with test requirements. Test requirements are entered to a test executor. The test executor runs test cases against the SUT and makes a report from the results. The executor is usually an external tool. [15; 16]

Offline MBT test suites can be stored and run anytime without regenerating the test suite. Therefore, it is possible to use the generated test suite for regression testing. When the program changes one only needs to change the model and regenerate a test suite. An offline MBT generator generates abstract test cases, which have to be made executable before running them. Test cases are made executable so that the generation tool writes tests in a format acceptable to the execution tool and the test execution tool then runs tests against SUT. Therefore tests are made executable partly in generator and partly in executor. The main thing is that performed test executions can be fully reused in the same test execution platform. [17.]

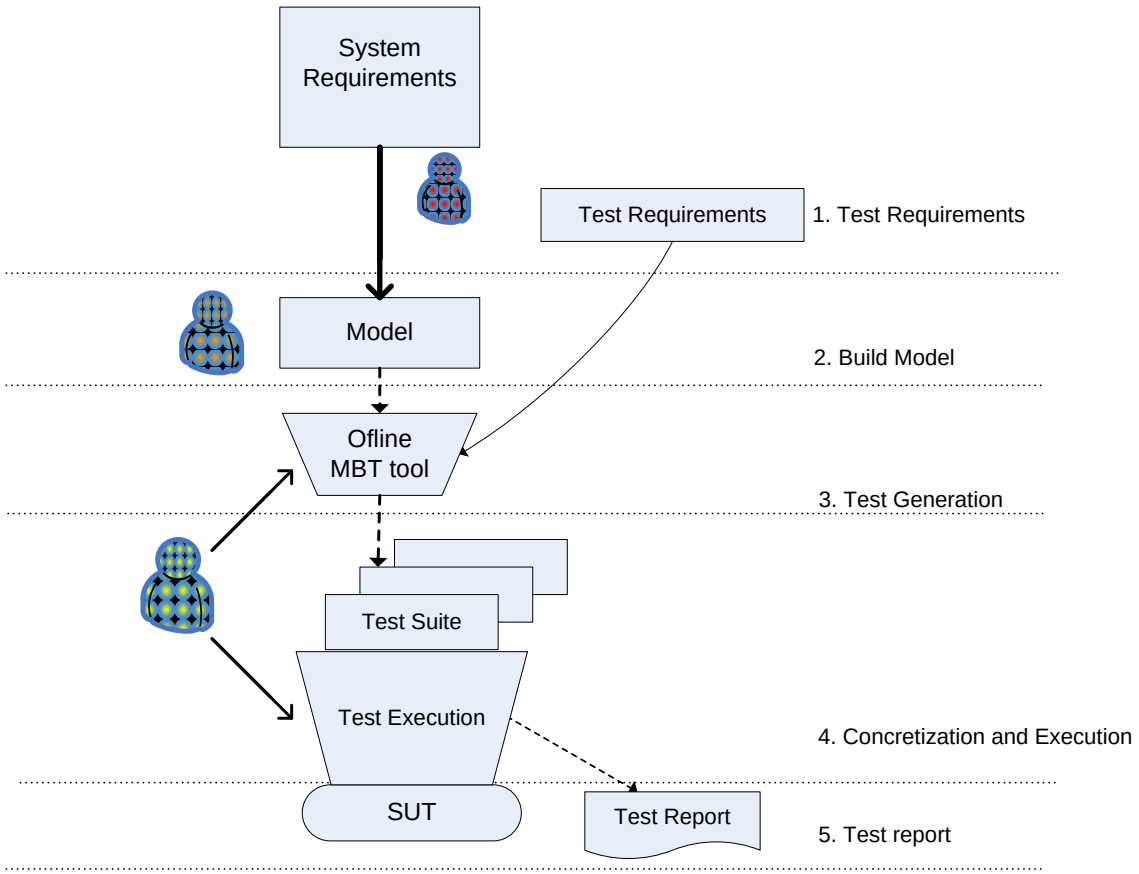

Figure 2.5: Offline model based testing approach, adapted from [6; 15]

2.4.2 Online approach

Model based approach employs online model based testing approach. It signifies that, UI level automation performed in this thesis will use online based approach of MBT.

With online model based testing, a model is created based on system and program requirements. Then the model and test requirements are imported to the MBT tool. In online MBT, a test generator and an executor are found in the same tool, because of the possibility to make tests generation and execution at a same time. Before online MBT can be started, the adaptation layer has to be implemented. The online adaptation layer joins the SUT and MBT tester together. When the designed model gets uploaded, both the test generation and execution is done by online MBT tool. Later, only after implementing adapter application, it is possible to start the test run. The online MBT tool performs test execution continuously after tests are generated, which means forwarding one-step in the model, running that step immediately in the SUT and analyzing the result. If the result differs from what it is expected, based on the model, the test fails. [17.]

Figure 2.6 describes the method of online model based testing approach.

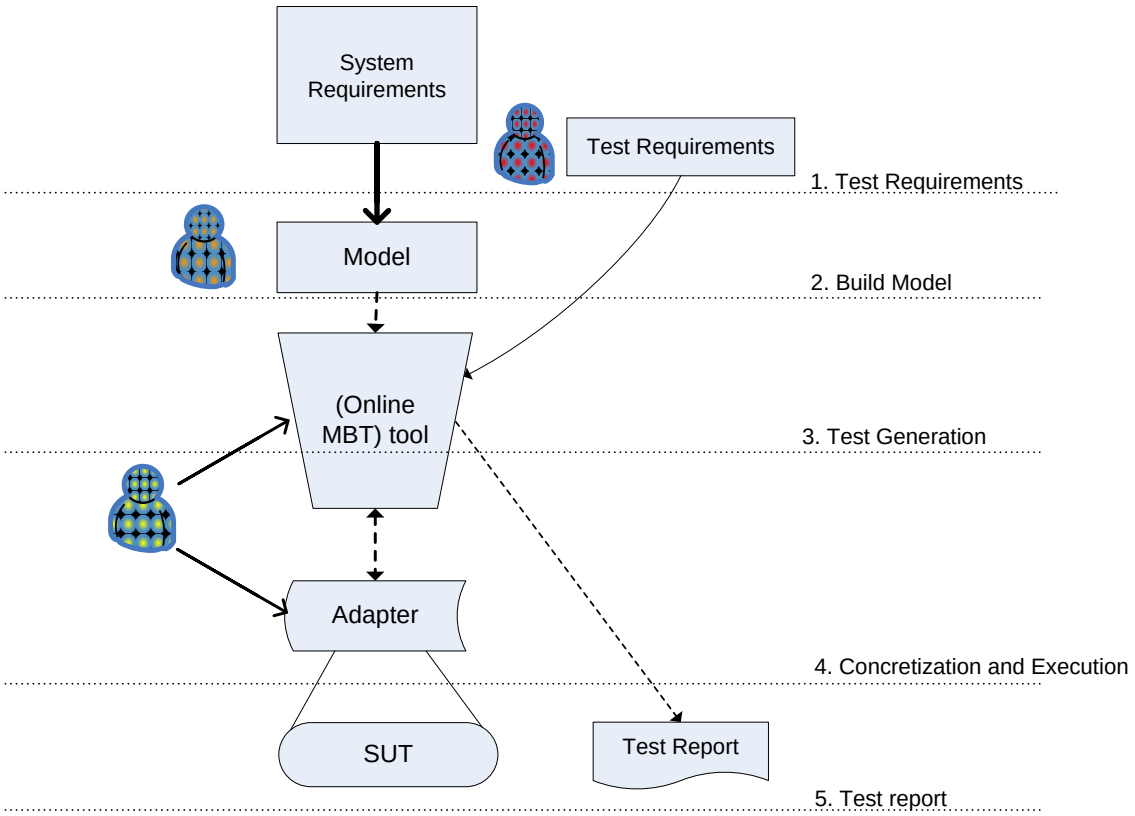

Figure 2.6: Online model based testing approach, adapted from [6; 15.]

Compared to the offline approach, the main advantages of online MBT are running infinite test suites and testing non deterministic systems. The online model based testing approach is connected directly and continuously to the model and this makes it possible to react continuously to changes and perform autonomous decision-making. Therefore, testing of non-deterministic systems is possible. By using online testing, it is possible also to make the testing session as long as required, or until the program crashes. This is especially useful when there is a need to test for example, memory leaks over a long period. [16.]

2.5 Potential Challenges and bottlenecks

Despite of numerous benefits of test automation explained in above sections, we often encounter very impractical and serious testing issues while executing the test cases. The manual testing plays its role in such cases. In today's context of testing, combination of both manual and automation in testing is inevitable to make sure that the errors do not run out of the grip and potential bugs can be hunted down.

Some of the challenges/bottlenecks with test automation methods are mentioned below [18; 19.]:

1. **Expert workforce needed**: It requires a special skill set to work with, write, and manage the test scripts. The people with these skills are often difficult to find and expensive to hire. They also need regular training to keep up to date with new techniques. This chaos may increase more when, there is only one expert in the team, and he is involved in different teams.
2. **Tools complexity**: Automation tools might possess some hidden defects and hence follows an incremental procedure of tuning. Moreover, there may be some hidden preconditions to setup these tools, which are not possible to communicate through installation directory or wikis. On such situations, creating automated testing scripts is very cumbersome and complex. It can take a team of people months and even years to set up properly.
3. **Maintenance issues:** Basically, automation in GUI testing depends very much on the way UI software has been designed. UI design keeps on changing every now and then, and developers will not finalize it until the best design is assured. Every time when the UI design changes, the scripts that were written to originally test the application have to be re-written for the changes. This is a time consuming task and can often take longer than manually testing the application in the first place.
4. **Resistive to change**: In practice, not all test combinations can be executed by the automation tools. There could be thousands of test combinations possible. And as explained earlier, UI design keeps on changing a lot, which forces the test engineers to change the script accordingly. Hence, to change such a large number of scripts in short time period continuously is not an easy task. There is always some limitation to the number and types of test cases to be selected.

5. **Reliability factor:** There are certain situations, where even in failure cases we cannot get any information from the logs generated by the automation tools. For example: while testing phone through automation tool, if device reboots or resets automatically, there could be numerous reasons behind it. And such critical behaviors could not be tracked by these automation tools. There we need manual participation to dig out the core issues, and try out other tracing methods.

Amidst the various problems mentioned above, the concept of test automation using model based approach offers several benefits in testing process. In short, we can summarize the benefits of MBT as follows. [19.]

1. **Easy maintenance**: All models can use the same test driver scheme to produce test script for the requirements captured in each model. When the changes in UI design occurs, only the logic of model needs to be changed, while when the test environment changes, the test engineer just modifies the test driver scheme.

2. Earlier and More Fault Detection: Model based testing not only automates test execution, but also automates test generation. In practice it means that the tests are generated and executed within same frame of time. This increases the chance of finding bugs in earlier phase already [20, p. 10]. With MBT, most of the bugs are found already in the modeling phase. Finding bugs in earlier phase helps developers to fix the bugs earlier too.

3. **Traceability:** Most of the MBT tools also provide the traceability from the tests to the requirements. This function makes the detection of the source of the faults easier. The test engineers can quickly find out the part causing the fault.

4. **Reduced Testing Cost and Time**: Using MBT tools, test cases generation and execution time can be reduced significantly. The requirements change only requires change in the model and that helps in saving a lot of time as compared to the manual design of the test cases.

This chapter described on how test automation works in general, how it differs in execution and where it needs to be addressed on some specific situations. Also, the importance of using MBT approach was explained. The upcoming part is centered on the implementation strategies, and more practical issues involved with deploying model based concept are explained.

3. UI level Automation: Smartphones

A smartphone is a mobile phone that offers more advanced computing ability and connectivity than a contemporary feature phone. [21]

Smartphones can be regarded as handheld computers integrated with a mobile telephone. They allow the user to run and preemptively support multitasking applications that are native to the underlying hardware. A smartphone runs complete operating system software providing a platform for application developers. Some examples of operating systems are Symbian, Android, Windows Phone, iOS etc.

This chapter describes on automation practices followed prior to thesis implementation. UI level Automation comprises execution of several test cases that are related to UI Software design. UI Software design for a smartphone is a broad field, and entails many components and applications inside it. For example: Camera, Messaging, Music (Audio, Video), Web Applications, TV applications etc.

This thesis focuses mainly on UI level automation based on camera specific to Nokia smartphones using Symbian S^3 OS.

The next section describes on goals set. Before delving deep into those, it is wise to have a look on what tools are being used with our approach. Following three independent tools are being used:

- **Testability Driver (TD)**: Ruby [11] based test automation tool using keywords/scripts owned by Nokia. The Linux version of TD is open source and free to use for development and testing.
- **TEMA**: Python and Java based GUI automation toolset for model based testing. Also an open source tool owned and licensed by Tampere University of Technology. This tool executes the model and makes it run with SUT.
- **TEMA-TD Adapter**: Making synchronization between above tools. Adapter holds logic of bridging a communication gap between these two automation tools.

More information on above tools is explained in upcoming sections.

3.1 Goals set

The goals set for this thesis can be summarized in reference to Figure 3.1. They are as follows:

1. Executing models on Ruby based automation tool (TD) directly, and automate the test execution (basically Camera based tests).
2. Use of TD in Linux for Symbian devices. (TD tool in Linux is developed for MeeGo devices only). Goal was to try using the tool for Symbian device, for example Nokia N8.
3. Executing same models in *Multiple SUTs* at a same time.
4. Communicate with SUT easily through IP address/WLAN.

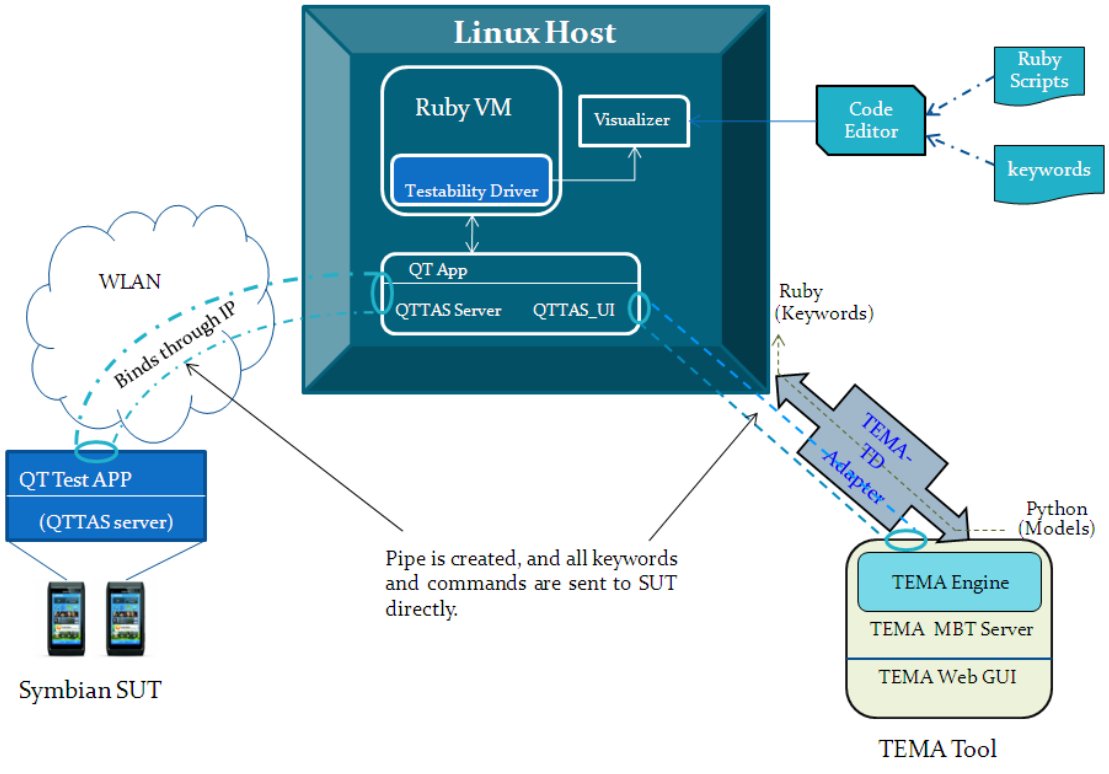

Figure 3.1: Automation Test Bed architecture

The test bed architecture shows the functional implementation of the test automation approach. The three tools mentioned in the beginning of the Chapter 3 are visualized in Figure 3.1. The goal is to use these tools and automate the software testing in SUTs directly. Before performing the test run, models are designed using Model Designer tool, and are uploaded to TEMA Web GUI. The coverage requirements for the test generation are also defined in Web GUI. Coverage requirements not only define the ending criteria for a test run, but also influence the test execution and the direction in which the execution tends in a given state. After having all the target roles defined and devices assigned, follows the test execution part. For this, Web GUI instructs TEMA test engine to initiate the test runs. TEMA test engine in turn listens to a port for a connection with adapter.

Adapter is an application that holds XML file for SUT definitions, and can check whether SUTs are ready for test execution or not. When SUTs get ready, the adapter application establishes connection with them and informs TEMA test engine. On receiving the connection information from adapter, TEMA test engine starts to run the server, which in turn executes the models. The phase of communication between adapter and SUT takes place through IP address generated by qttas server application running in both SUT and host. SUT when ready gets connected to WLAN. Afterwards, qttas server running in SUT generates IP address connection details. This IP address information is stored in XML file inside the host, which is fetched by adapter before a test run is started.

3.2 Tools used

This section talks about the automation tools involved in more descriptive terms.

3.2.1 Testability Driver (TD)

Testability driver is a testing tool open sourced and owned by Nokia. It has been used for automation purpose, basically with Qt applications running on any platform that runs Qt. Platforms that have been successfully used are: Linux, Windows, Mac, Symbian, and MeeGo. [22.]

The basic architecture of Testability driver as shown in Figure 3.2 is explained below.

1. Language
Ruby language is supported as script by Testability driver. Ruby is expressive, and easy to learn quickly.

2. Agent
Agent is the component that runs on the SUT and handles the communication between the applications and testing framework.

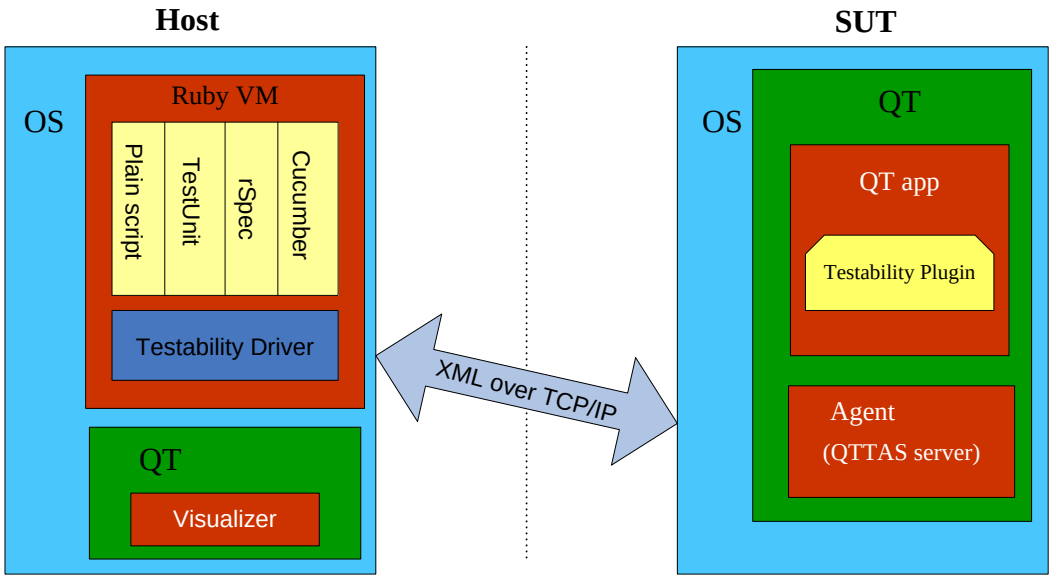

Figure 3.2: Testability driver Architecture, adapted from [22]

3. Testability plugin
This library will be loaded by started applications. This will give access into process of the tested application.

4. Communication
Agent communicates with testing framework using XML over TCP/IP address. XML files contain the information regarding SUT and the type of communication being used by SUT to connect to TD. Several methods like USB, Bluetooth, and IP address can be used for communication.

5. TDriver ruby library
This is a class library implemented in Ruby language and provides access to communicate with the target SUT in Ruby language.

6. Visualizer
Visualizer is an application that visualizes the application under testing. It helps to find out the objects in the application and also the properties for each object. The Visualizer application with a SUT's home screen captured is shown in Figure 3.3. In Image view section, the view in SUT gets captured to Visualizer. On the right hand side, there is more information about the type of objects being opened in Home screen. Lower part of Visualizer contains code editor, where scripts can be written and executed in SUT.

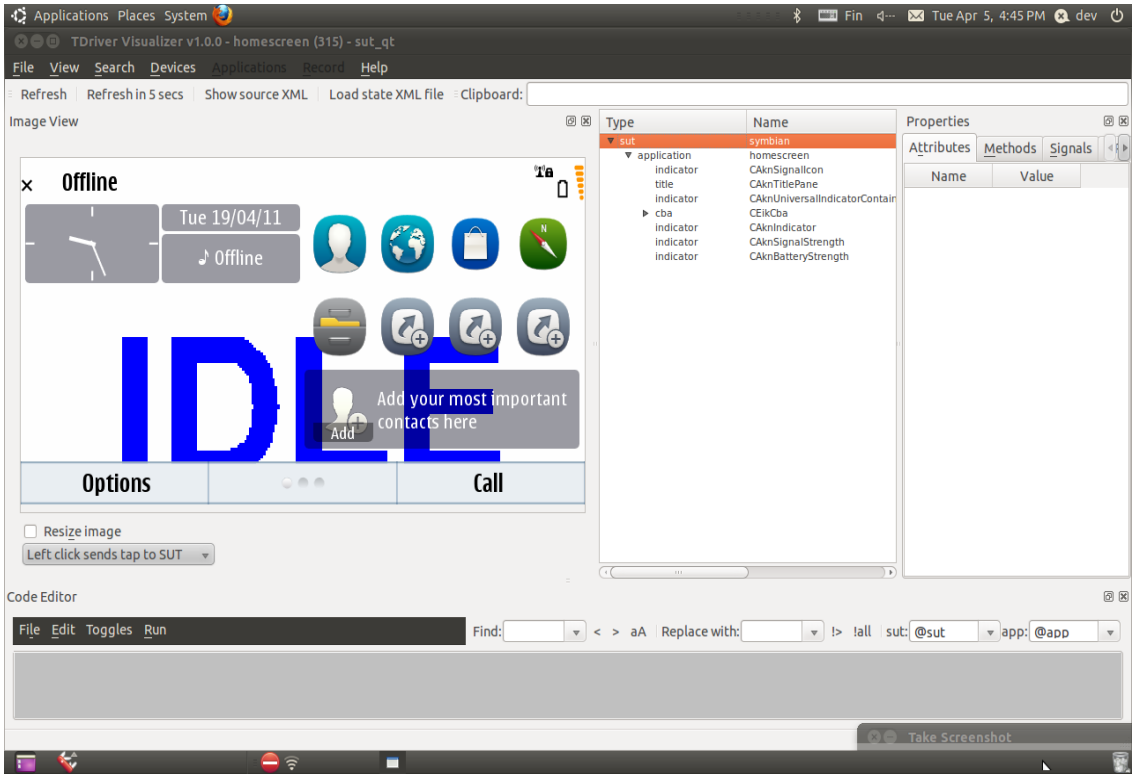

Figure 3.3: Visualizer mapped with SUT's home screen view

More information about Visualizer will be discussed in Section 4.4.

3.2.2 TEMA Toolset

TEMA Toolset is a package targeted for easier deployment of MBT in the domain of smartphone application GUI testing. This toolset is developed and owned by Tampere University of Technology, Department of Software Systems. The methodology is based on long-term research on MBT and practical case-studies with industrial partners. The features of TEMA's two-tier modeling approach include the ability to reuse high-level models as the basis of test generation among different smartphone platforms. [23.]

The practical product of TEMA project is a set of tools designed for creation and execution of model based tests. The toolset can be divided in five distinct parts, and its structure is illustrated in Figure 3.4 inside a dotted box.

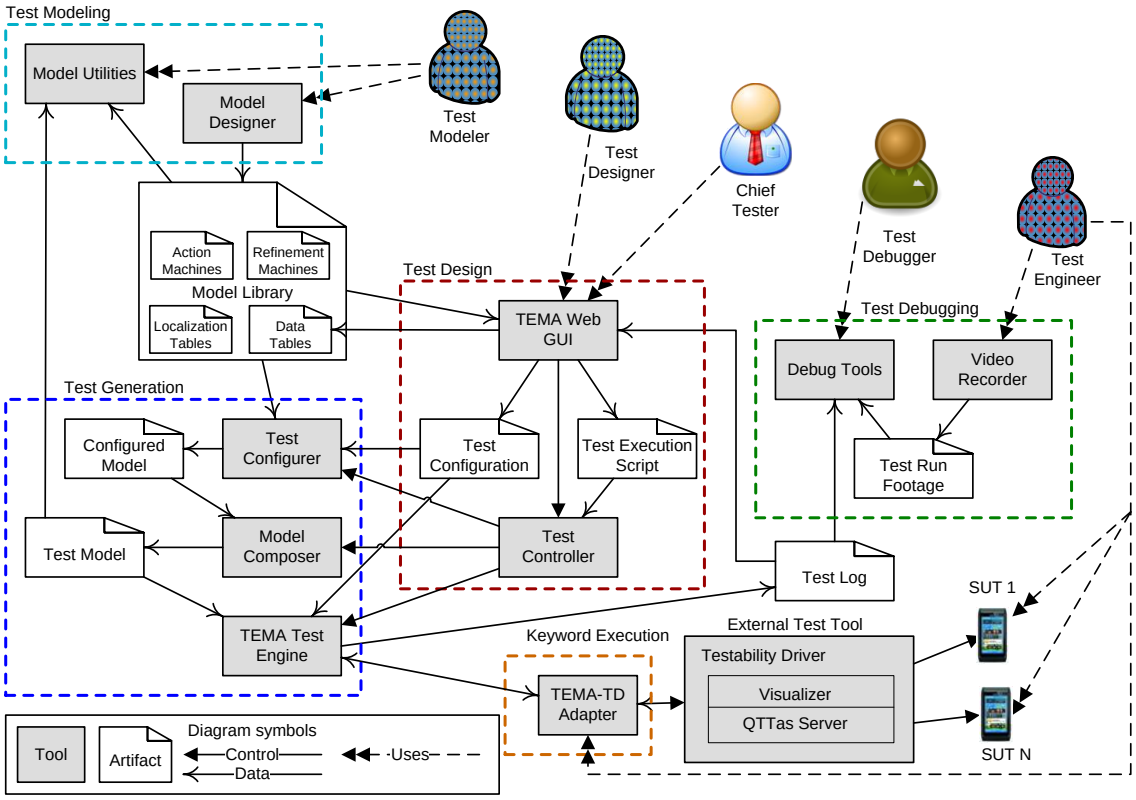

Figure 3.4: Test tool architecture, adapted from [24]

The first part is Test Modeling where models and its corresponding data tables, localization tables etc. are created. Second part is Test Design and Control where tests are launched and observed. The third part consists of Test Generation that is responsible for assembling the tests and controlling their execution. Fourth part is keyword execution which holds the logic of binding SUT and Engine, and communicates them with help of keywords. Fifth part is Test Debugging, which deals with analyzing the test log generated after test execution.

Test Modeling is done with a design tool, *Model Designer*. It is a tool for creating action machines, corresponding refinement machines, and data tables. Action machine is a model component that describes the functionality of the SUT at the level of action words. Similarly, Refinement machine is a model component that contains *keyword* implementations for action words. Keywords are low-level GUI events, used for implementing action words. [20, p. 11]

For example: the event of launching camera, if categorized to model based components, will look as follows:

Action word: *aw_LaunchCamera*

Keyword: *kw_LaunchApp 'cameraapp.exe'*

Data table is a data structure containing the external data to use in data statements. Whereas localization table is a data structure that contains localization data. Thus the output of Model Designer in first part is basically a well balanced model where required steps of automation are recorded in terms of states, in structure of Finite State Machine.

After model designing part, next is to design how to control the testing of these models. For this, it contains a Web GUI which is used basically to launch the test runs. The step on setting up the test is to specify a coverage requirement which defines what must be done in order to complete a test. After that, other parameters of test are set, such as number of SUTs, number of adapters used, types of SUT, as well as the algorithm to be used in test generation. Combined, the coverage requirement and the other parameters can define very different test runs, from executing use case to running the test case randomly as stress testing. All this information is sent to the test generation part, which starts running the test. As the execution proceeds, all significant events will be captured into a test log. The Web GUI observes the log and provides a real-time feedback on the test run.

After setting up test run through Web GUI, a test controller instructs test engine to initiate test generation. For this, test controller first checks the coverage requirement it received, from Web GUI and determines what model components are required for the test run. These are passed to Model Composer, which combines them into a single test model. This test model is handled by test engine, which determines the next steps based on parameters received from test control. Both test control and test engine report the progress of the test run into a test log. [24, p. 18]

When test engine starts to execute the test run, basically keywords gets executed through models. Test engine relays them to the adapter application, and waits for the response on connection to SUTs. The adapter in turn, checks the XML file for SUT definitions and establishes a connection with SUTs. The adapter tool not only converts keywords into the form understood by the SUT, but also manages the gradual execution of complex keywords and returns data on whether the keyword execution was successful or not back to test engine. For this thesis, we have used two Symbian based SUTs, and these SUTs in the first hand are known only to adapter.

Practically, we had an external test tool called Testability driver, which is basically a keyword based automation tool using Ruby Language. TEMA, on the other hand being a MBT tool, was having totally different technical implementation. On such situations, the role of adapter becomes crucial. When the models get executed completely, test log can be downloaded from test engine, where the information of keyword execution and their status are recorded.

4. Methodology: Building Automation Test Bed

As discussed in the earlier chapters, this thesis implements the model based testing concept with two different test tools and one adapter application in hand. This chapter will show the methodology followed in building a test bed structure. Building a test bed structure here mainly implies use of a Model Designer, Web GUI, test engine and SUTs along with external test tool; Testability driver. In addition this chapter will put more focus on screenshots of the components and tools used in the test.

4.1 Test Modeling and Execution Environment

Models creation and their execution are important phases of model based testing concept. In short, Model Designer tool allows creating a model and the execution of model is accomplished by using TEMA Web GUI. From our thesis scope point of view, we will focus mainly on the implementation part with the help of GUI design. A model targeted for two SUTs will be created, followed by an explanation of action machines and refinement machines implementation.

4.1.1 Model Designer

Model Designer, as defined in Section 3.2.2, is the primary model creation tool in the TEMA toolset. A few of its tasks are allowing the creation of model components and data tables to be used for test automation, management of the model library, generation of the utility components required in model composition, and assembly of the components for test runs.

A GUI design of Model Designer is shown in Figure 4.1. The upper left part of the figure shows the domain under which the product has been created with. The name Symbian refers to the domain in the context of Figure 4.1. The lower left part of the Model Designer contains a section that displays sequence of actions and corresponding attributes used while creating action machine and refinement machine designs.

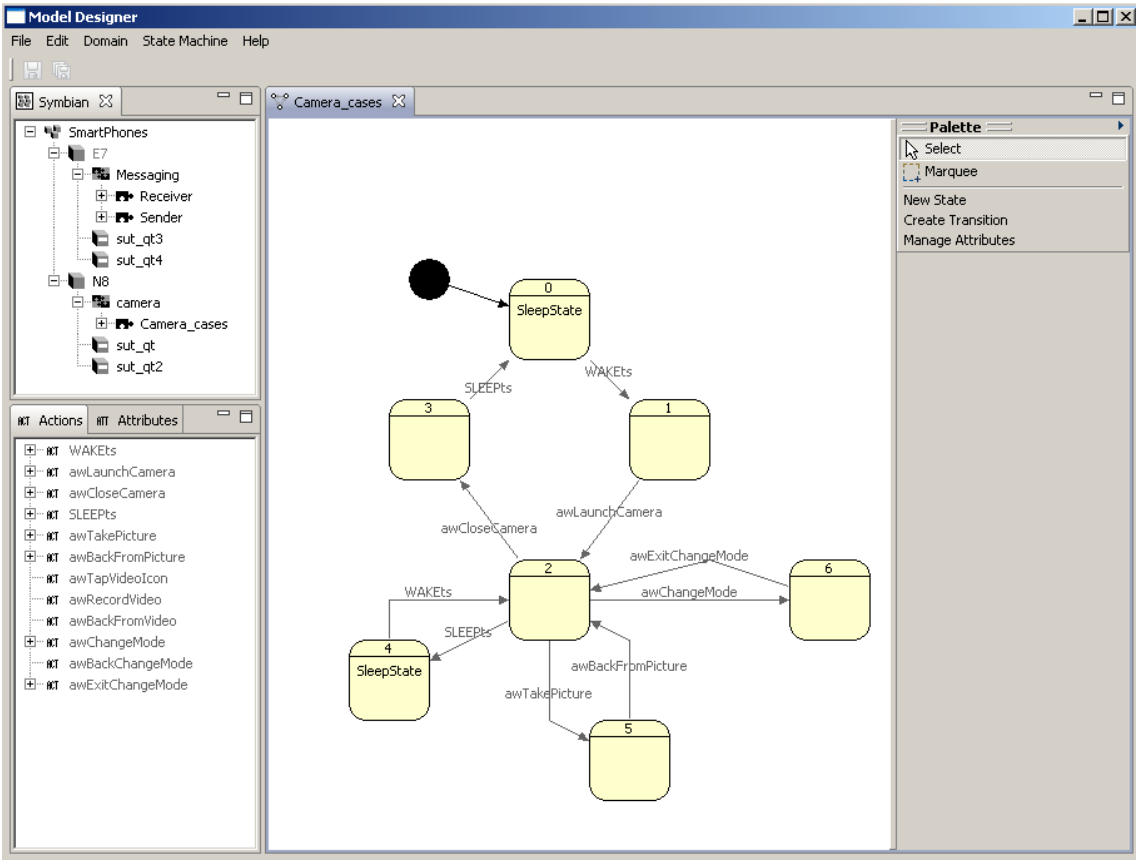

Figure 4.1: Model Designer UI design

The center part of Figure 4.1 shows the action machine implementation. Before discussing on action machines and refinement machines, it is wiser to first see how Model Designer tool is used to create a new model package. The procedure goes as follows: A domain is created first, followed by the product family. Inside product family we can have one or more products depending upon the requirements. In our case, we created two products inside a product family. Similarly after creating a new product, we can assign a new Concurrent unit. Inside each Concurrent unit, there are action machines and refinement machines. Figure 4.2 shows a structure of the domain created with Model Designer.

A point worth noting in Figure 4.2 is the SUT definitions. There are two different SUTs being used namely *sut_qt* and *sut_qt2*. They differ by unique *id* value. More explanation on SUT definitions could be seen from Chapter 5 of this thesis, where SUT being used in test runs will be shown along with the underlying details.

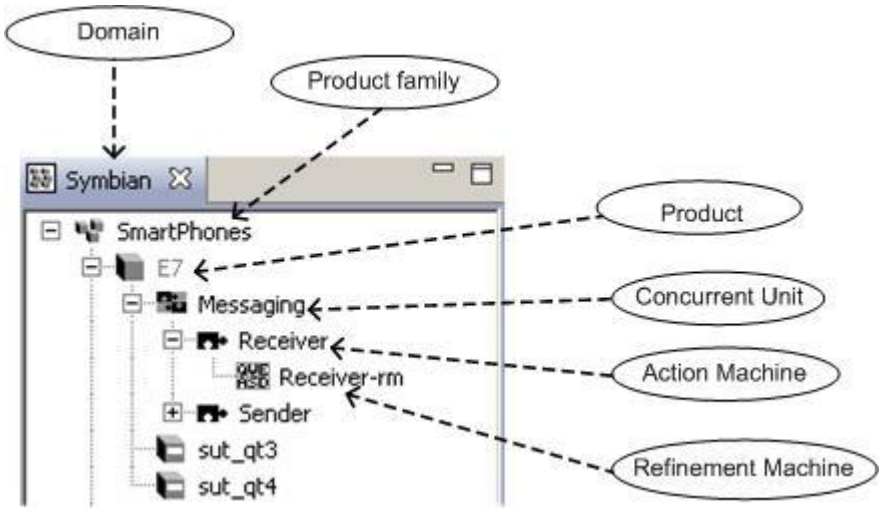

Figure 4.2: Symbian Domain and its structure in Model Designer

As an important component of a Model Designer, we have a sample action machine and refinement machine created. As shown in Figure 4.3, each action started with **aw** is basically an action word. Action machine holds the execution logic of the individual events recorded inside a test case, and thus sequences followed in action machine implementation are strictly followed during execution. In practice it means that action related to *CloseMessaging* will start only after messaging application is opened. And this execution sequence is held by action machine.

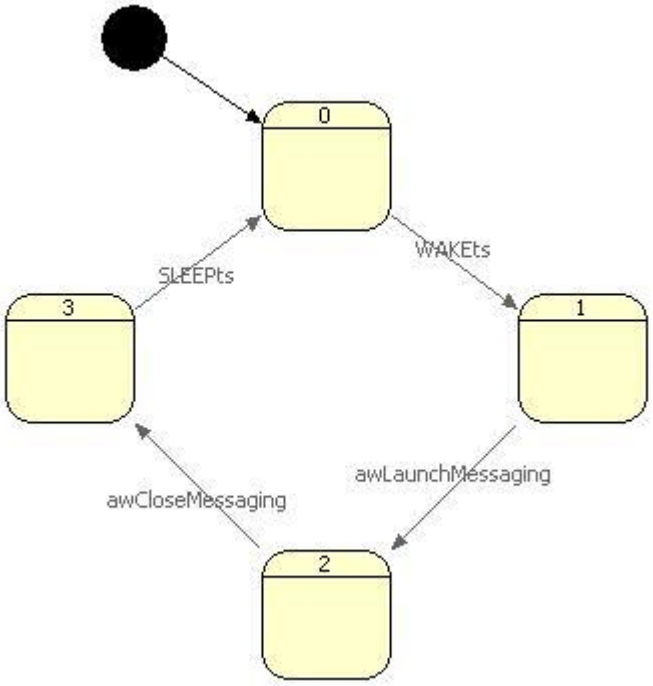

Figure 4.3: Sample action machine design

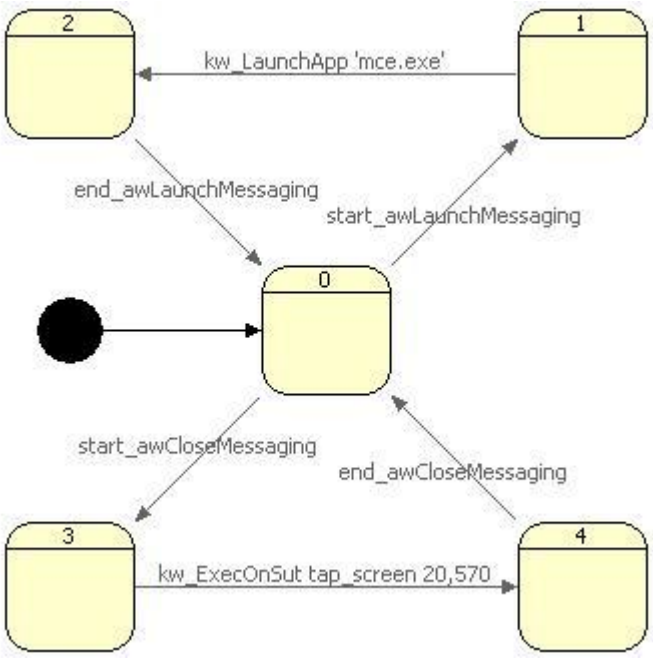

Figure 4.4: Sample refinement machine design

Refinement machine on the other hand deals with keywords, and possess detailed explanation on each action words associated with action machine. For example: while executing model package, when action word *awLaunchMessaging* is witnessed, it switches to corresponding refinement machine implementation and initiate the set of action to be executed for that action word.

As shown in Figure 4.4, when action word *awLaunchMessaging* is invoked, it is implemented in refinement machine with three sequences in row: *start_awLaunchMessaging*, *kw_LaunchApp 'mce.exe'*, and *end_awLaunchMessaging.*

The refinement action that launches the messaging application and is understandable to SUT is the keyword *kw_LaunchApp 'mce.exe'*.

Thus after creating a model package bundled with requirements in terms of state, the next step is executing these model followed by the test generation phase. This initiation is carried out by TEMA Web GUI. We will discuss the role of Web GUI next.

4.1.2 TEMA Web GUI

Within TEMA tool set, Web GUI holds the responsibility of launching the test runs, when model package gets uploaded. Before the launching of test runs, Web GUI needs to follow the sequence of activities, for example: the loading the model package, selecting the test mode, defining target roles, data table selection etc. Web GUI also checks whether other parameters of the test are set, such as number of SUT, number of adapters used, types of SUT, as well as the algorithm to be used in test generation.

At first, a coverage requirement must be specified in order to define the desired areas to test in the test run, i.e., what must be carried out to complete the test run successfully. Figure 4.5, shows the way of specifying coverage requirement through the mode selection in Web GUI. Typically, the coverage requirement is a logical expression composed of actions that are interconnected with logical operators such as 'AND', 'OR' and 'THEN'. The order of executing these actions, action words and keywords in practice, can be further modified with parentheses. However, the coverage language also admits of the presentation of coverage requirements in the form of regular expressions, enabling the execution of aforementioned long-period tests by, for instance, executing all actions of the test model, resulting in a virtually endless test run. [20, p. 14]

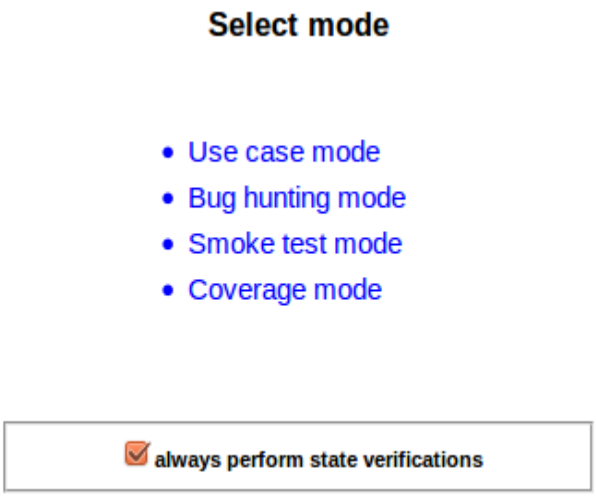

Figure 4.5: Coverage requirement through mode selection

In Figure 4.6, two SUTs with different device definitions which were created in model package are being assigned the target roles, before pushing them to test runs. Also the number of adapters running for the automation purpose makes sense. For our case, we had only one adapter running for translating keywords from model to SUT. Once the device assignment is successful, device settings are saved, and Web GUI reaches to the phase of test run. All these saved contents initiating test runs are recorded in test log, which can be downloaded after test run is finished.

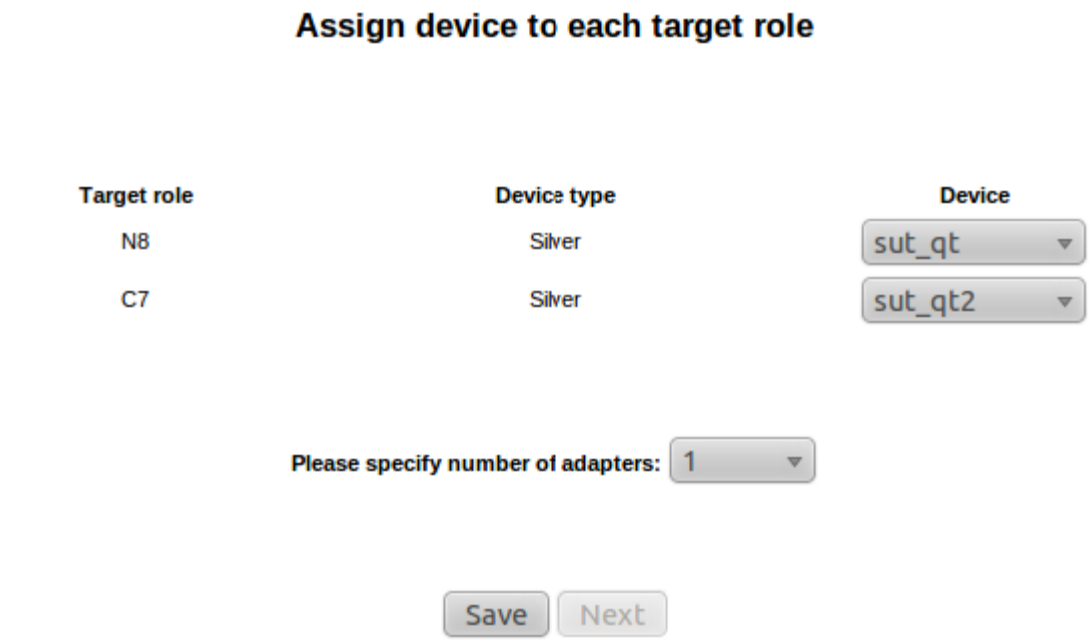

Figure 4.6: TEMA Web GUI assigning role to two different SUTs

Figure 4.7 shows Web GUI launching a test run. It can be seen that there is a long list of actions that Web GUI performs before initiating test runs. There is a choice before commencing test run regarding display of executed events. For example, if you want to see only keyword related events, you can simply check the *show keywords* box.

Figure 4.7: Web GUI launching a test run

4.2 TEMA Test engine

After a test run gets started, test engine plays a central role in the generation of tests. But before that, the system checks if a test configuration is created successfully or not. If a test is successfully configured, it goes on composing a model successfully. Next is to execute the generated test run in SUTs. At first test engine sends a query to adapter. Since, adapter holds a file with SUT definitions, whenever SUTs get ready, adapter can be used to bind these SUTs ready for execution. Figure 4.7 shows adapter waiting for a connection from clients.

Once adapter realizes SUTs to be up and running, it creates a communication pipe between test engine and SUTs directly. The moment adapter gets connected with a Client (SUTs), test engine becomes active and test execution gets started. As shown in Figure 4.8, every time when a test engine executes the tests, it is verified by adapter to check if the keywords get executed to clients (SUTs) or not.

After the test run finishes or when it is stopped, the log file can be downloaded from the GUI. This log will contain the detailed information on different events execution.

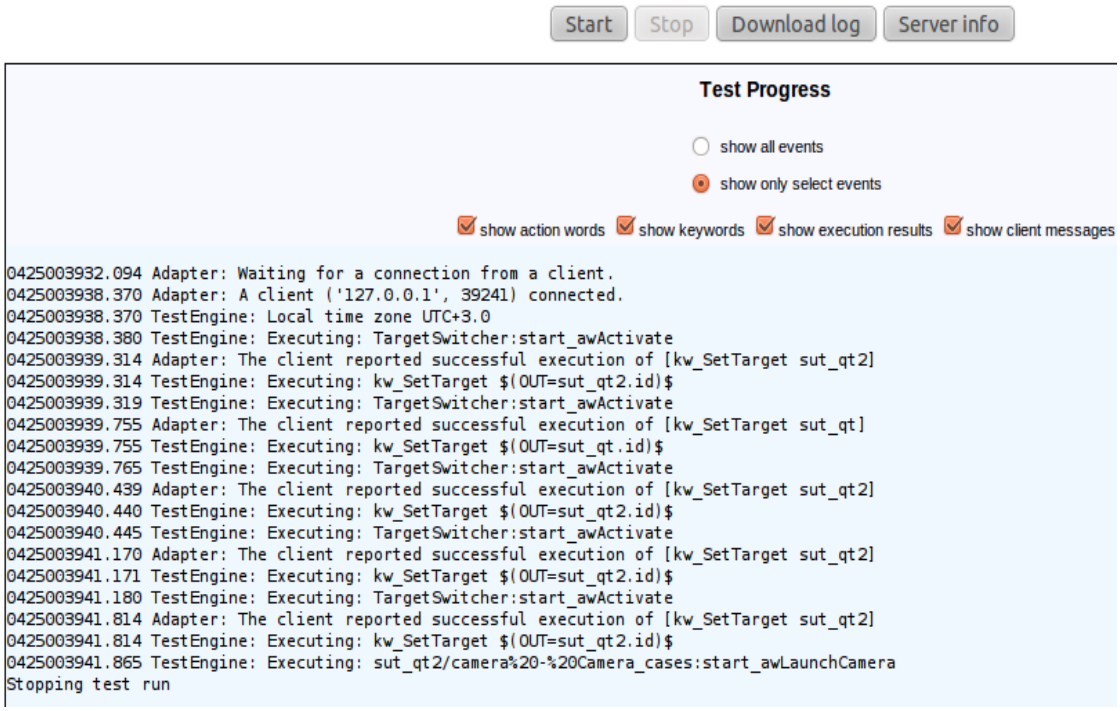

Figure 4.8: Test engine executing keywords on SUTs

4.3 TEMA-TD Adapter

In the course of the test run, keywords are executed in the test model and these keywords are further relayed to the SUT by test engine. Adapter plays a role to translate the keywords in between and verify that the SUT has successfully executed them. Furthermore, the adapter tool not only converts keywords into the form understood by the SUT, but also manages the gradual execution of some more complex keywords and naturally returns data on whether the keyword execution was successful or not back to test engine. [20, p. 18]

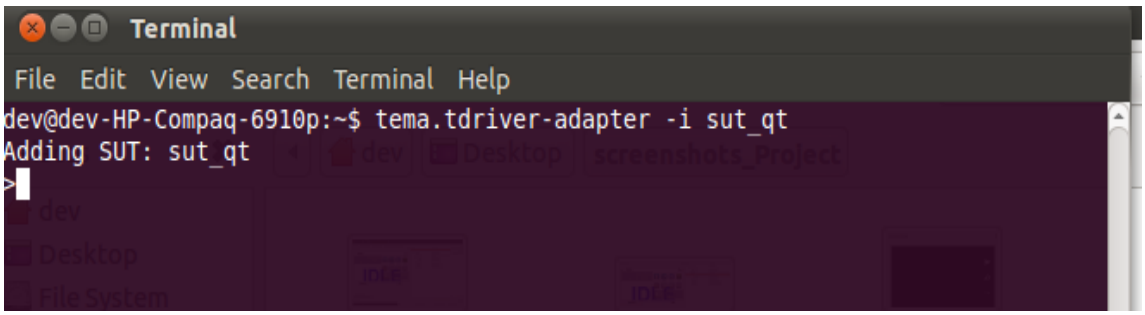

Figure 4.9: Adapter initializing SUT

The name TEMA-TD refers to the test tools, and adapter communicates with SUTs to bridge the gaps of these tools. Figure 4.9 shows adapter adding the SUT to check if SUT is ready and running. The sign > shows that SUT was added successfully with device name *sut_qt*. More SUTs can be added the same way, just by specifying different device names in parallel.

4.4 TD Visualizer and SUT

Visualizer is purely a TD based tool. It is used basically to show the SUT to the user in a similar fashion as TD perceives it. Visualizer shows how a SUT is composed of test objects and where particular test objects can be found on the UI. Also attributes, behaviors, methods and Qt object API are shown. Visualizer also helps scripting by providing an UI for creating attribute based object identification strings. Visualizer consists of three main parts: Image view, Object tree, and Properties window. [26.]

Image view part is responsible for *capturing screen view* of SUT being connected at certain point of time. If there are more than one SUT being used, then SUT that is selected as active connection will be mapped in Image view. In Figure 4.10, Image view part is situated on upper left part of Visualizer, and messaging application has been mapped in Image view for *sut_qt*. This is because *sut_qt* is selected as an active connection, and the messaging application of a SUT is launched.

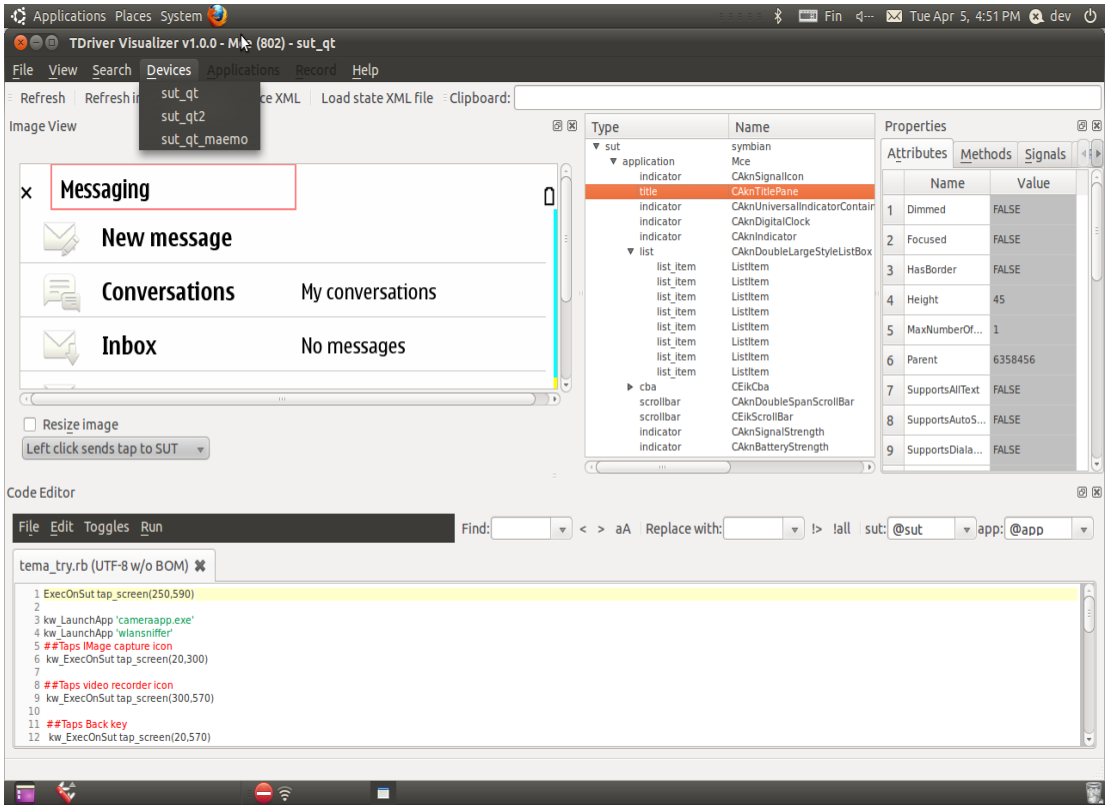

Figure 4.10: TD Visualizer components interacting with SUT

Object tree, situated in the upper middle part of Visualizer, depicts the hierarchy of GUI objects of the SUT. Selecting an object in the tree will highlight it in the image view. Right clicking on items brings up a context menu with further options.

Properties window on the other hand shows a list of objects and their types. This list contains only those objects which are currently being opened with active connection in Image view. In the upper rightmost side of Visualizer, the properties window also shows more details about the selected object in tabs, including Attributes, Methods and Signals for Qt SUT. The Methods tab shows the proper semantics of using attributes while scripting with TD.

The Visualizer also includes a ruby code editor that will help on writing and fixing automated tests. Code editor section lies on lower half part of Visualizer. The purpose of Visualizer code editor is to integrate test script coding and SUT inspection into one application. Having built-in code editor allows TD-specific features, as well as inserting data from SUT into editor directly without using clipboard. [27.]

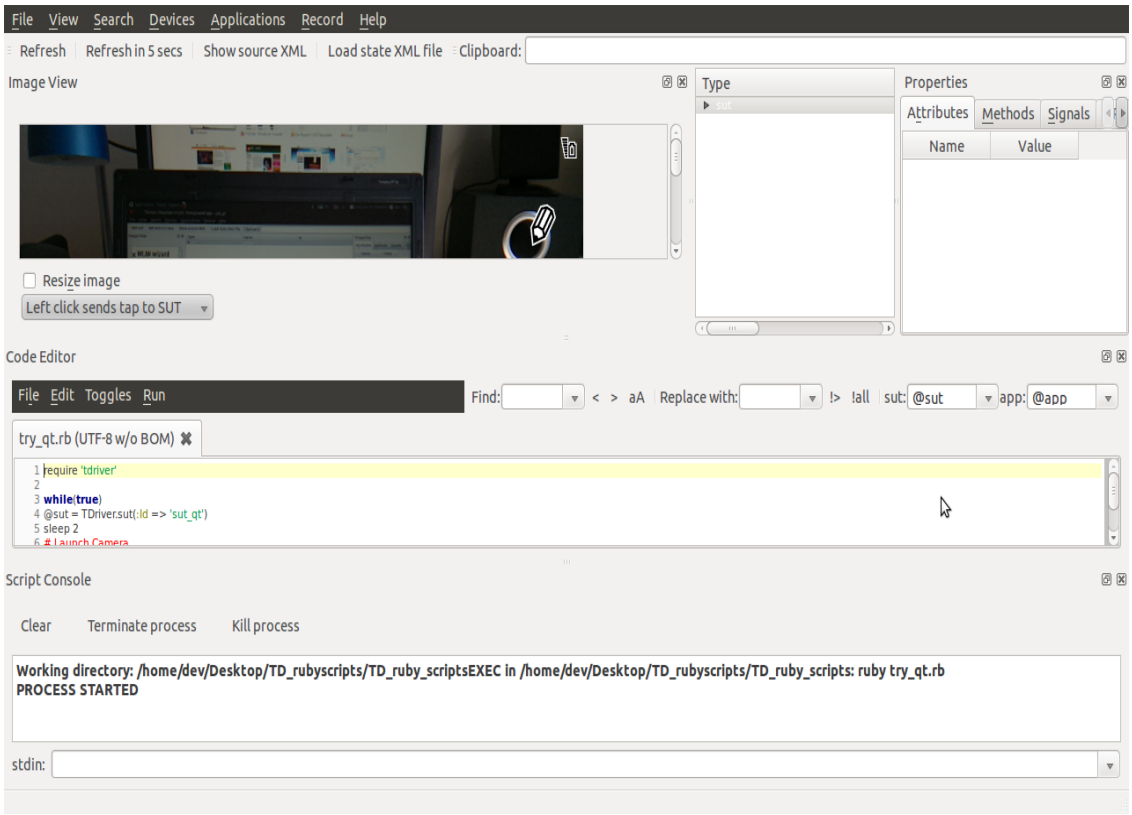

Figure 4.11: Code editor executing script

Figure 4.11 shows the code editor starting to run the Ruby script. When a script is run through code editor, a script console appears which shows the progress of test script and reports failure if some problem occurs during script run. While script is being executed, the Image view in Visualizer shows the screen capture in Figure 4.11. More information on installation of Visualizer and script in Ruby with TD can be seen at [27].

5. CASE STUDIES

After building test bed for the model based implementation and having tested the connection to SUT through adapter in interactive mode, the challenge was to be able to execute the entire model (which basically comprises several keyword implementations) on SUT. For this purpose, we chose two real test use cases to automate. One was related to Camera, where Image capture and Video recording were automated for a long period testing. And another was related to messaging, where multi-phone messaging activity was automated. This part of thesis will contain explanation on specific procedure followed such as: SUT definitions, adapter implementation, action machine design, refinement machine design, and TEMA test engine implementation.

5.1 Case Study I: Image capture and Video recording

Automating the use cases of Image capture and Video recording incurs series of test steps. The decomposed steps are mentioned below.

(A) Image capture use case possesses following finer steps:

(i) Launch Camera.
(ii) Press Capture button or tap capture icon.
(iii) Press or tap 'Back'.

(B) Video recording use case possess following finer details:

(i) Launch Camera.
(ii) Tap on Video recorder icon.
(iii) On video mode, press capture button or tap capture icon.

With above mentioned test steps, it is clear that the model based implementation needs to automate these steps and it should be executed successfully in SUT(s). In other words, these steps are the user requirements that are supposed to be automated. Thus, the first step is to design a model through a Model Designer tool that can incorporate all those test steps in one bundle.

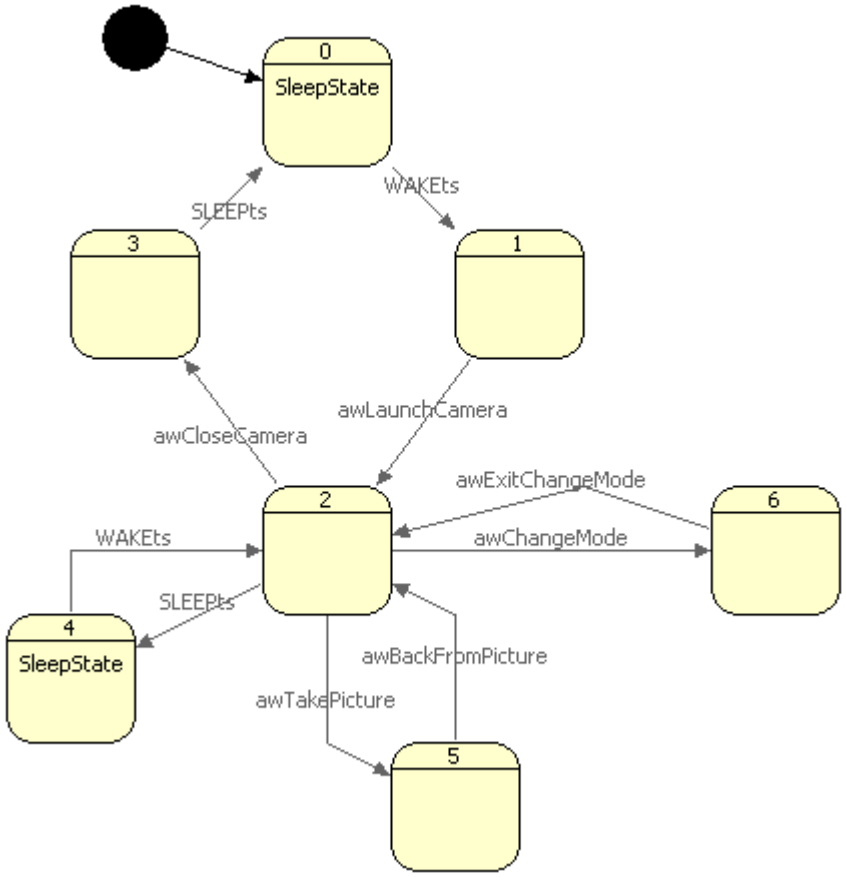

Figure 5.1: Action machine design using Model Designer

Designing a model here mainly includes designing of action machine and refinement machine. Figure 5.1 shows the action machine design that possesses execution logic to automate the image capturing and video recording steps.

Action machine contains series of action words starting with a suffix *aw*. Each action words are meaningful in the sense, whenever the test engine witnesses any action words, it processes the action and the results are seen in SUT. Figure 5.1 shows that execution of *awLaunchCamera* results on launching the camera, and *awCloseCamera* closes the camera. As mentioned in Section 4.1.1, action machine also tells about the order of execution that the test steps should follow. For example: action word *awTakePicture* should be executed only after successful execution of action word *awLaunchCamera*, and that sort of logic is set with action machine design. Next is refinement machine design, which actually deals with keywords and behaves according to action machine design.

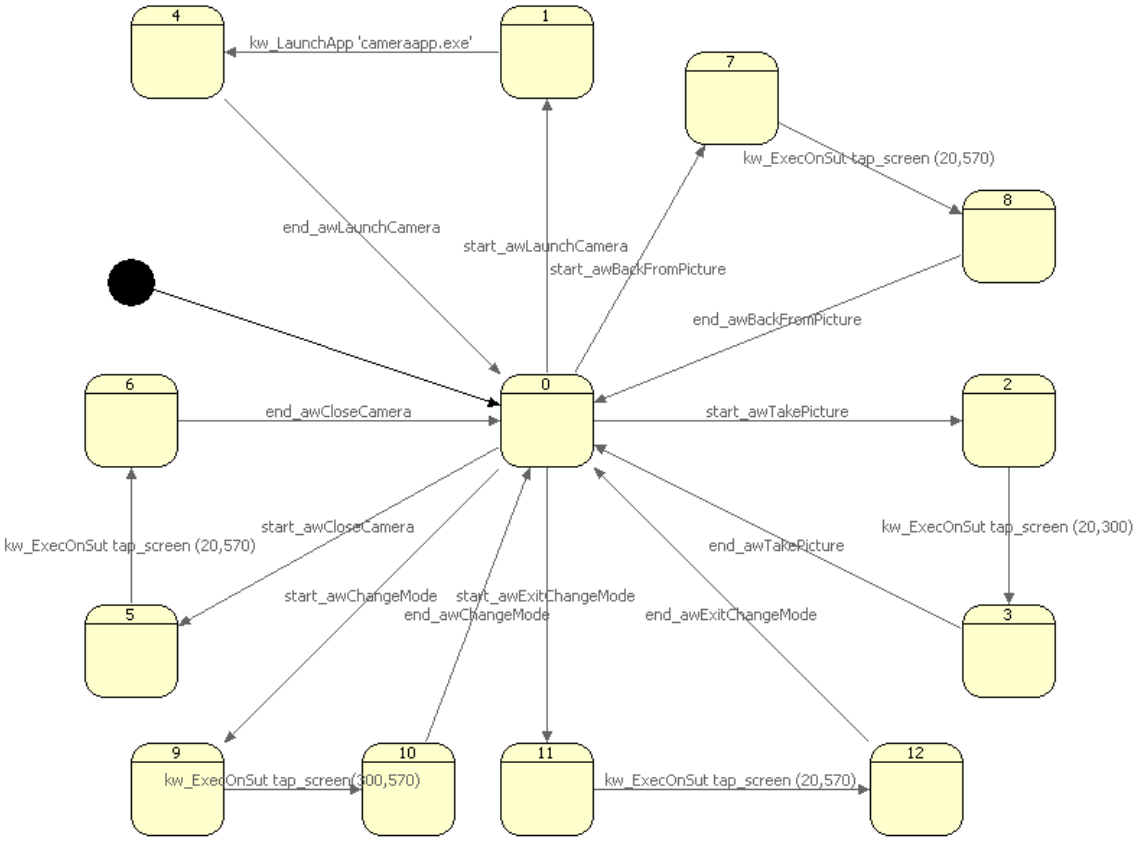

Figure 5.2: Refinement machine design using Model Designer

As shown in Figure 5.2, refinement machine basically elaborates each action corresponding to action words. Refinement machine specifies the order of keyword execution corresponding to each action words. Hence, implementation of refinement machines becomes straight forward if we have robust action machine design. On the other hand, the figure shows states where transitions are implemented using specific coordinate values. For example: the keyword *kw_ExecOnSut tap_screen 20,570* has been used in specific for certain transitions. This coordinate values can change rapidly with each new SW releases, and in such cases the new models needs to be created again. This can simply increase the number of states exponentially high, which might be difficult to maintain in the long run. For our thesis scope, this problem can be overcome by using a suitable adapter application, which enhances the possibilities of using wider range of keywords. Based on this an efficient model can be designed, that requires less maintenance and is not affected by frequent UI changes in SW releases. Section 4.1.1 showed some more explanation of sample refinement machine design.

After designs are ready, a model package comprising these designs in bundle is extracted and fed to Web GUI, to get ready with test execution and generation. Web GUI is a tool that enables launching of test runs. Before launching test runs, a series of activities are performed. At first the extracted model package needs to be uploaded through Web GUI. Figure 5.3 shows a glance of how uploading is done.

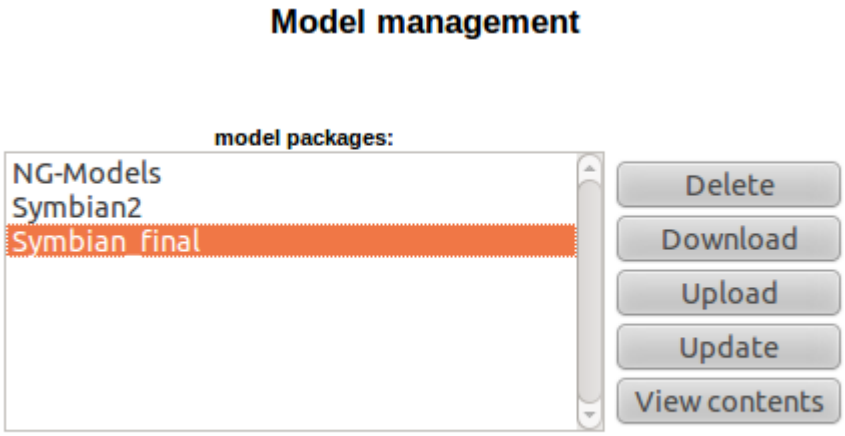

Figure 5.3: Model package uploaded through Web GUI

There could be multiple model packages in the list, each containing different model implementation. In our case, the name of model package extracted was *Symbian_final*, which contained the model design for automating both the Imaging and Messaging use cases. After successfully uploading the model, a coverage requirement must be specified in order to define the desired areas to test in the test run, i.e., what must be carried out to complete the test run successfully. Section 4.1.2 can be referred to see how the coverage requirements are set through mode selection.

Following the mode selection phase, the next action that Web GUI performs is *defining the target roles*. Figure 5.4 shows method of defining target role to a single product.

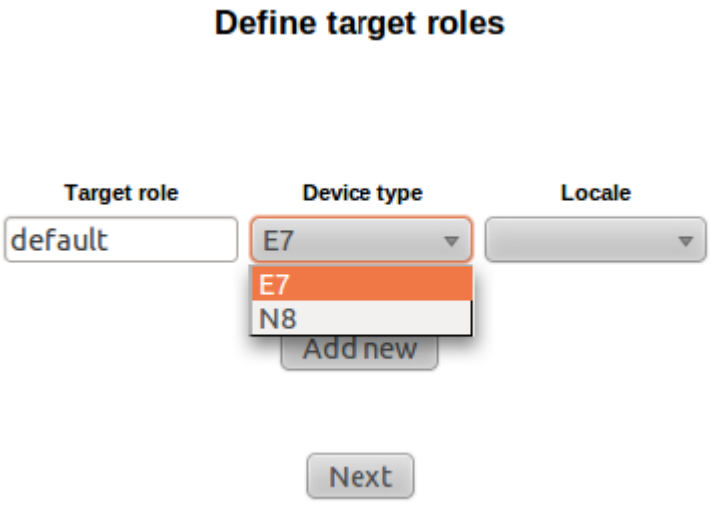

Figure 5.4: Defining a single target role to a product

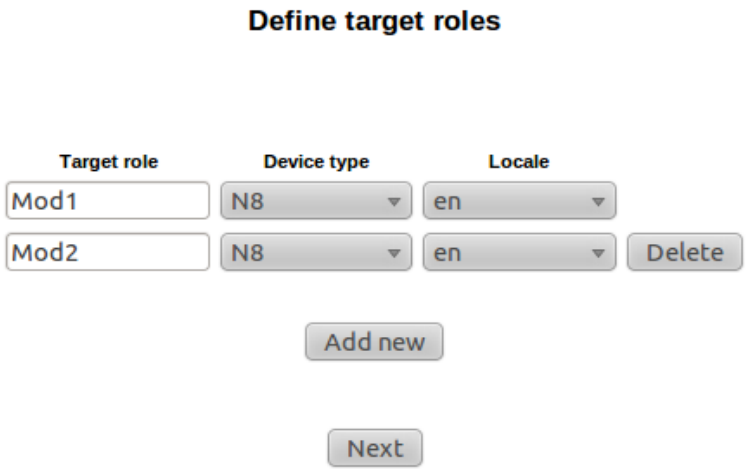

Figure 5.5: Defining multi-target roles to products

This involves assigning a certain device type a specific target role. In our case, we had a model created for two products, namely N8, and E7. N8 was modeled for Image capturing and Video recording purposes, whereas, E7 for messaging automation.

Web GUI on the other hand can also accommodate more target roles, if more than one products needs to be tested. In this case, we add one more target role, and select a different device type.

From our thesis scope point of view, we will assign two target roles to the same device type, so that there is uniformity in execution. For example in Figure 5.5, the device type N8 has been chosen for two different target roles.

Main goal to perform this action is to see whether one model can be executed to more than one device at the same time or not. Also with Web GUI the need of creating two device types arises because we have two SUTs implemented for automating Image capturing and Video recording cases namely *sut_qt* and *sut_qt2*. Each of the device type created can be assigned to any of those SUTs. Next is to select applications to each target role, as shown in Figure 5.6.

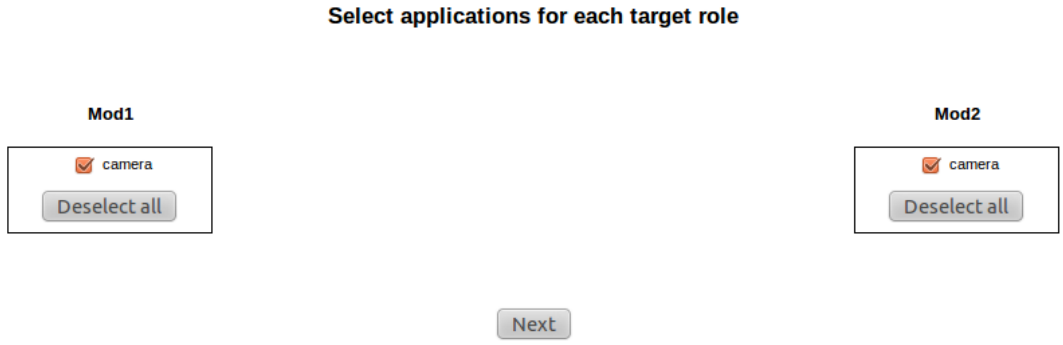

Figure 5.6: Selecting applications for target roles

Only camera application is seen in Figure 5.6 because model package designed includes only camera application implementation for product N8. After selecting applications for each target roles, next step is to assign a device to each of the target roles so that the selected application could be run on the assigned device. Figure 5.7 shows the method of assigning a device to each target role.

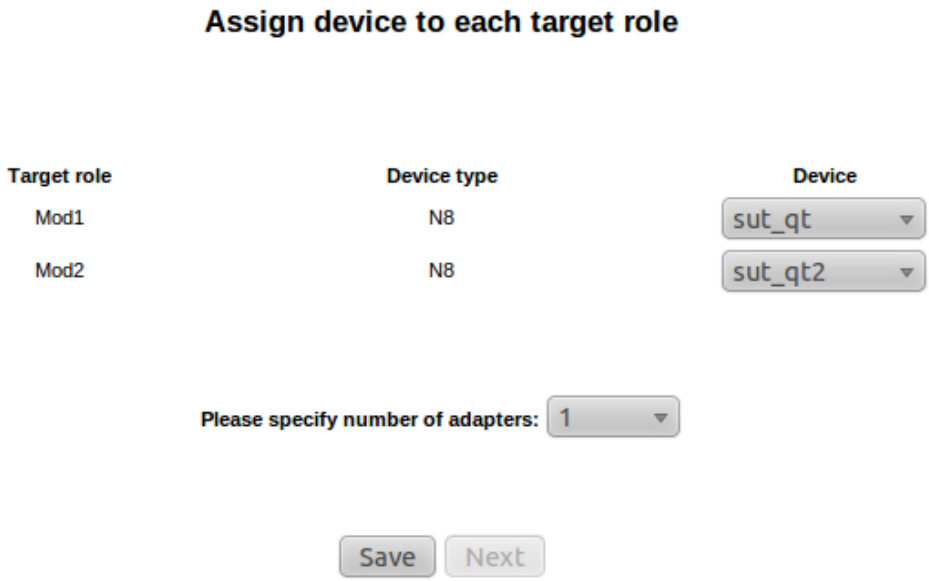

Figure 5.7: Device assignment to each target roles

In the Figure 5.7, the two different target roles have been assigned to two devices, *sut_qt* and *sut_qt2*. After successful device assignment, the Web GUI prepares the test configuration package from the series of actions performed earlier. This test configuration package is fed to the test engine, and the test engine initiates the execution of test models in real. And to implement test execution in real SUT, we need to make sure that the SUTs are up and running.

But, there is a barrier in communication between SUT and test engine, as they are implemented under different technical variations. Hence, we have used adapter to bridge the technical gaps between test engine and SUT. Adapter can track and verify that SUTs are up and running, and in other hand test engine can track adapter.

Hence, following three actions must be verified to get succeed in executing models in real.

(i) SUTs are defined in XML definitions in Host.

(ii) Adapter checks and gets connected with SUTs.

(iii) Test engine executes, and finds adapter connected to SUTs.

Figure 5.8 shows XML file used for SUT definitions. This file holds the definitions for one SUT, in this case *sut_qt*. This XML file resides in Host side, and is accessed by adapter later when the connection to SUT is needed before test execution. Host keeps

connection with SUTs through IP address. For that, both SUT and host has qttas server running in common. When SUT is ready to be tested, the qttas server is started in SUT, and it generates IP address on the screen. This IP address is unique, and does not match with any other IP addresses.

```
<sut id="sut_qt" template="qt">
<parameter name="qttas_server_ip" value="192.168.1.35" />


<!--<parameter name="input_type" value="touch"/> -->

<parameter name="kEscape" value="0x04"/>
<keymap xml_file="keymaps/qt.xml" />

</sut>
```

Figure 5.8: SUT definitions in XML

If more SUTs are to be tested, each of them has to have qttas server running so as to generate IP address and get ready to be connected to host. The IP address generated in individual SUTs must be recorded in the file as shown in Figure 5.8. To include more SUTs in test runs, we need separate definitions with unique Id (for example: *sut_qt2)* and unique IP address.

Next, adapter will check if the SUTs are connected or not. Figure 5.9 shows an adapter getting connected to SUTs.

In Figure 5.9, adapter successfully adds *sut_qt* and *sut_qt2* in the connection list. Also, using adapter console, keywords can be executed directly in interactive mode. This is useful in the first round to test whether adapter connection with SUT was really successful or not. After successful binding of adapter with SUTs, test engine execution becomes meaningful. Now we can track back to the above mentioned three actions back to back. First, SUT definitions are ready. Second, adapter gets connected to SUTs already. Now, the only action needed is test engine starting the test runs.

With first and second actions being successful, execution of test engine goes straight way. That is, when a test run is executed by test engine, it first checks adapter regarding SUTs connection. Figure 4.7 in Chapter 4 can be referred to visualize how the test engine waits for adapter connection with SUTs. Since, adapter is already connected, test engine starts performing test execution, and as a result the keywords that were modeled in model package start to execute in real SUTs.

```
dev@dev-HP-Compaq-6910p:~$ tema.tdriver-adapter -i sut_qt sut_qt2
Adding SUT: sut_qt
Adding SUT: sut_qt2
>
```

Figure 5.9: Adapter connecting with SUTs

Figure 5.10: Test engine executing model package contents

Figure 5.10 shows test engine successfully executing the models. It shows the console of Web GUI, consisting of series of events executed by test engine along with adapter's verification of SUT connections. When the execution ends, either successfully or with failure, it stops test engine, which needs to be started later again to resume the execution process. Test engine records details of execution in a log file, which can be downloaded after the end of test execution. The generated log file is shown in Appendix section. Nokia E7 and N8 were tested as a sample SUT for this model test execution as shown in Figure 5.11.

Figure 5.11: SUTs (Nokia E7 and N8) used in Test Execution

It is important to mention here that to execute the same model in different SUTs; the SUTs should have same baseline software. For example, it means if a model for image capture is designed for a product N8, then to be able to execute the same very model also in another product (e.g. E7), there should be similarity in GUI design related to camera operation. Support for multiple SUT execution with a single model not only helps on faster bug finding, but also saves time, and repetitive test runs can be made easily. And especially the Nokia's smartphone including N8 and E7 use similar camera UI designs. Hence, reusability factor grows much higher.

5.2 Case Study II: Multi-phone Messaging

Multi-phone messaging is one of the popular use cases tested in UI level automation to ensure whether a phone can handle multiple tasks concurrently or not. Basically idea is to automate the process of sending SMS and receiving it successfully. Hence, basically the design of this use case needs at least two SUTs, one working as sender, and the other as receiver. For this case study, we will only discuss on designs of action machine and refinement machines. Rest the test launching and execution mechanism is similar to earlier case study. That is after model package is ready; the three actions mentioned earlier needs to be fulfilled prior to initiating test run which involves role of test engine, adapter, and SUT definitions.

Figure 5.12 shows the structure of a multi-phone messaging model comprising individual action machine and refinement machine design for both sender and receiver. Designing a model for sender requires relatively more steps in comparison to the receiver. The reason is because the sender is involved in many different actions like writing a text message, allocating the receiver's name, and initiating the sending process. Whereas, receiver simply receives the text message and verifies it. This verification message is recorded in log and/or also possible to see in Web GUI console.

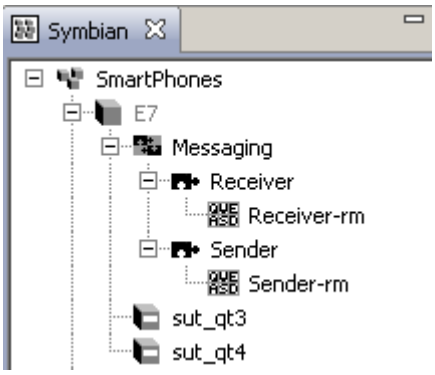

Figure 5.12: Structure of a Multi-phone Messaging model

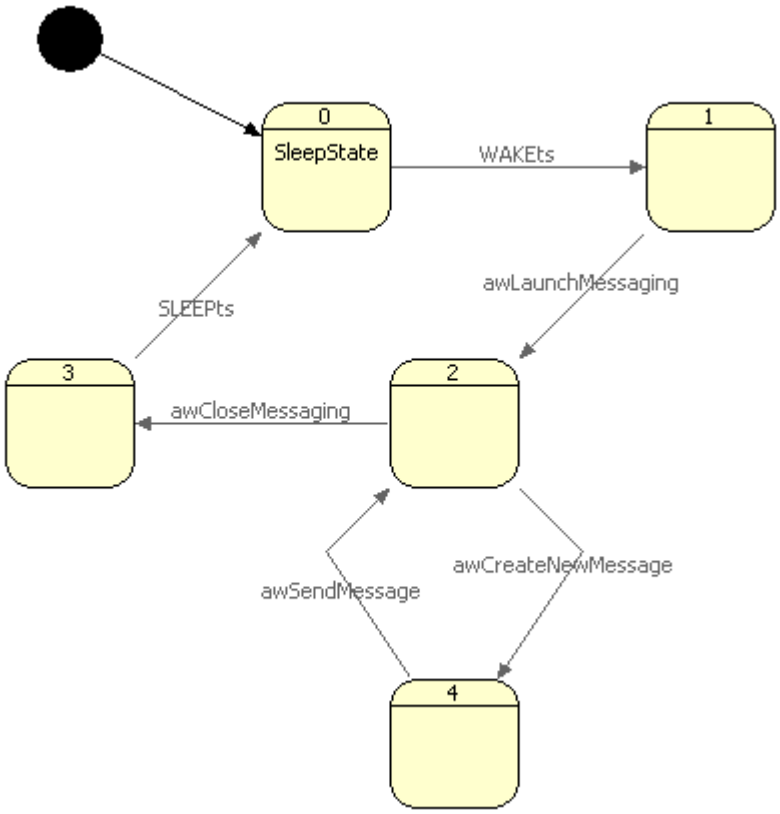

Figure 5.13: Action machine design for Sender

Action machine implementation is shown in Figure 5.13. This design simply contains the upper level action words that tell about the logic of messaging to be followed. This is refined into more detailed form in Figure 5.14 with the refinement machine design.

Figure 5.14 shows refinement of action words into more detailed form. This refinement action will be targeted to only one SUT that has been chosen as Sender. In this case, *sut_qt3* has been chosen for sender role. Hence execution of the model will first target the sender, and in sender side message gets typed, composed, and sent to the receiver. Receiver on the other hand, upon receiving a message, launches the inbox and opens the message. In this case, *sut_qt4* is the receiver.

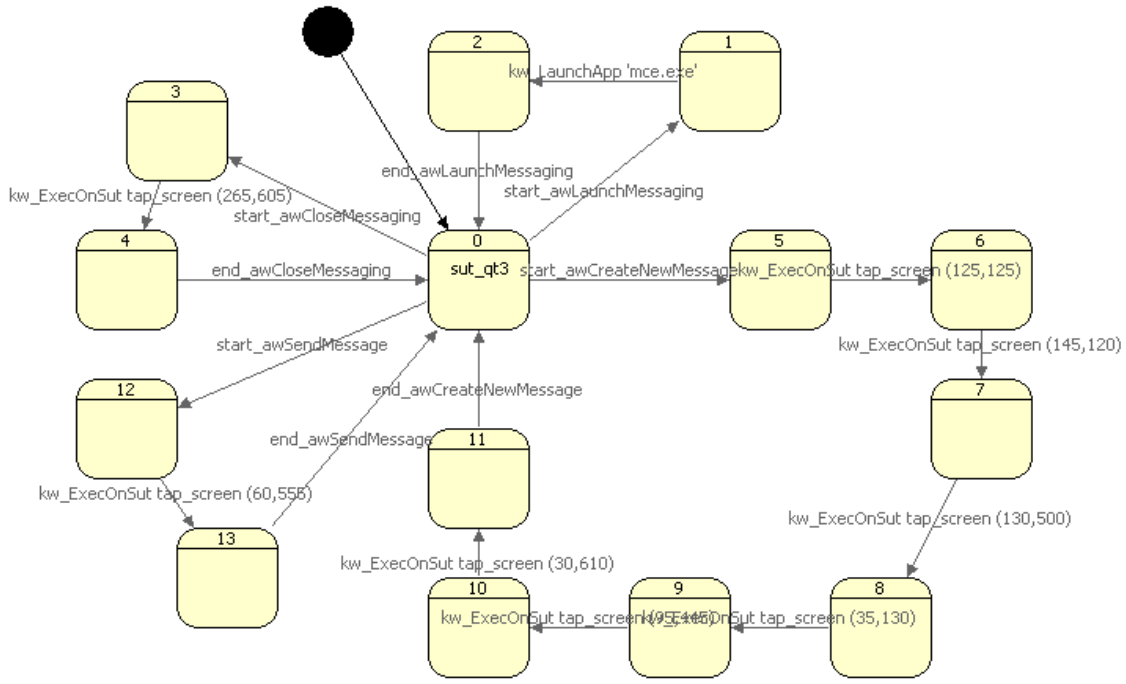

Figure 5.14: Refinement machine design for Sender

Thus, the implementation of action machine design for receiver contains action words involving launch of messaging and opening message inside phone inbox as shown in Figure 5.15.

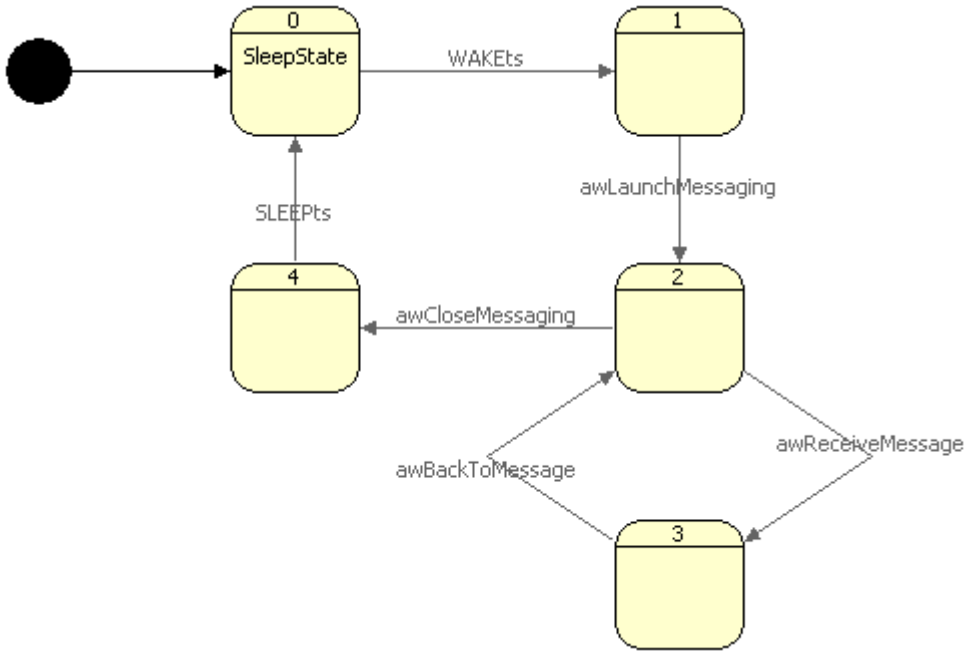

Figure 5.15: Action machine design for receiver

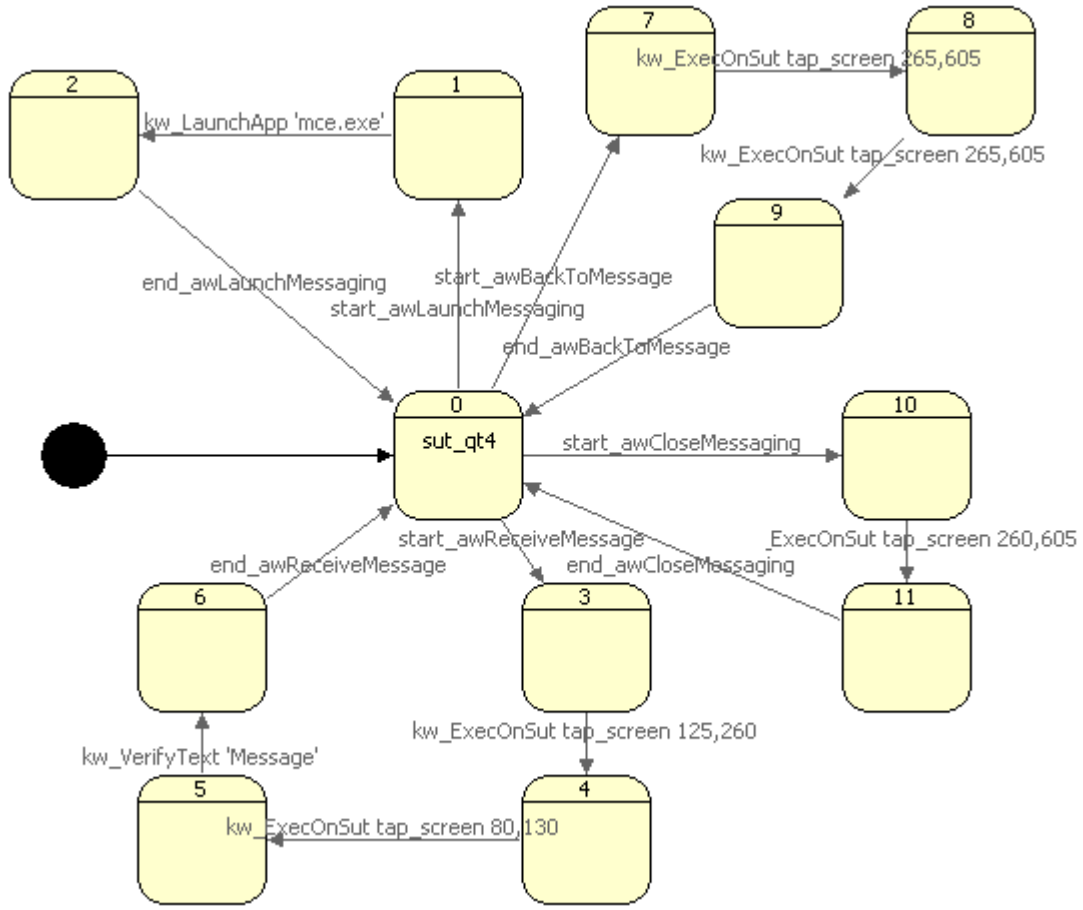

Figure 5.16: Refinement machine design for Sender

Refinement machine for receiver will hold the logic to implement the action words in the above figure. Action word *awReceiveMessage* is refined further into smaller actions. It consists of actions like opening inbox, message, and then verifying back about message details. Figure 5.16 shows the refinement machine implementation for receiver. The design of implementation is quite similar to sender, only verification part is bit different.

5.3 Analysis of Results

The two case studies performed can be used as a reference to evaluate the goals set in Section 3.1. The analysis of test results is based on how well the test methodology mentioned in Chapter 4 addresses on achieving the goals.

The main goal achieved was the successful execution of a single model to multiple SUTs at the same time. In other words, we were able to execute one or more use cases on two different phones simultaneously. The entire test run performed were divided in three different combinations. First combination included only testing of camera based actions.

Second combination included only messaging related tasks. And with third one both combinations were tested jointly.

Camera based test run automated the actions like image capturing and video recording in a loop. The test run successfully captured 1000 still images and around 800 videos in three hours. No problems were seen during test execution, and test run went smoothly. Similarly, messaging related test run automated the text message sending procedure. One of the SUT composed the text message, and sent to the other SUT. The other SUT checked message and sent back the received confirmation. This test run basically focused on the use of parallel testing. The most productive test combination included both camera and messaging in a single test run. The choice of selection for executing either of them was random. Most surprisingly, the execution of these two different test combinations went fine.

Another important finding of the case study was implementation of connection method between SUT and adapter using IP address. In general, when such automation practices are carried out, we make use of connection methods like mini USB cable, Bluetooth etc. But, these connection methods might cause chaos when there are more SUTs to be tested and you have limited Mini-USB cables, or suppose if the host does not support the Bluetooth connection at all, then it could really be difficult. We have used IP address settings as connection method in our implementation, which simply requires WLAN settings as default. When SUTs got connected to WLAN, the qttas server was activated which in turn produced unique IP addresses, and these IP addresses were assigned to individual SUTs during test execution.

Similarly, with this case study we have successfully tested the interoperability between two different test tools. TEMA tool which is basically responsible for model design and execution worked with a keyword driven test tool TD. In fact, SUTs that were executing models are aligned to work only with TD and related keywords according to their origin. Also, an implementation that we achieved through the case studies was use of TD in Linux for Symbian Devices. TD tool in Linux is mainly developed for testing MeeGo devices only. We were able to use this MeeGo based test tool successfully for Symbian devices. With this implementation, the entire goals that we planned were achieved.

Despite of many such benefits in automation practices with TEMA toolset, there are still some areas where improvements could be made. One such area is model package management. When multiple model packages are uploaded, sometimes the deletion of certain uploaded models is not possible. The system throws an error saying that the models cannot be deleted and it is being used by the test engine in some way. This issue of deletion is not a blocker of test runs though. The reason is because it does not affect the performance of the system and test execution at all. But definitely the fix for such issues are expected in the upcoming releases of TEMA toolset.

Moreover, with the case studies we have seen relatively small models in use, where test execution with TEMA stands good. But in large scale automation phases where multiple tests runs needs to be carried out, where size of models can grow huge; the

overall performance of the test execution might be of greatest concern. On the other hand, there are enough evidences of TEMA being used for executing much complex and large cases too. One such use of TEMA toolset can be seen from [24, p. 44-46]. This kind of extensive long period testing is a regular practice followed in companies these days to find out the bugs.

6. Conclusion

Model based testing, though being a promising test automation methodology, has not been favored yet in mainstream automation strategies in companies. The reasons could be several, from inherent difficulty of adopting completely new ideas in organizations to the lack of suitable tools for designing test models and generating tests. From our case studies it is quite evident that online approach to model based testing could be a very useful tool for carrying out long period testing, parallel testing and testing related to memory leaks. Also the maturity in test modeling possessed by Model Designer looks quite promising. This has definitely put model based testing in front row and justified the reason behind its popularity.

From the company's perspective, model based testing with TEMA could be a good candidate tool for industrial adoptions, provided that certain modifications are done in the existing system. A few of those possible modifications are mentioned here as recommendations. These recommendations are purely based on the experience gained during the thesis completion. First and foremost, the installation of TEMA toolset should be easier and straight forward. During installation of this toolset there were many dependencies to be followed strictly, and it had a fair chance of user getting lost into details. It would be good to have a single setup file that consists of all the needed information, and one time installing should do the work. Another challenge to model based testing approach is the complexity of a Model Designer tool itself. One needs to follow the documentation and rules strictly before starting to use Model Designer. If the task of designing model is simple and understandable, the extra cost to manage the modeling part can be minimized. It is because model creation and execution could be efficient if the testers possess some skills of test scripting and automation beforehand. Thus, it signifies that the plain testers to perform model creation need trainings and it can be tedious job sometimes to find such people. Another possibility to solve this issue could be assigning a separate role of 'test modeler' inside a workgroup.

Similarly, models themselves have also some drawbacks. The biggest one of those is the explosion of state-space needed. Even a simple application can contain so many states that the maintenance of the model becomes difficult and tedious task. This issue could be very critical, as it can have an impact to the stability of the tool itself. Thus, more robustness could be achieved if we can overcome the issues of maintainability. Finally, an adapter application should be written effectively. Effective here signifies that a test modeler should be able to utilize maximum set of keywords implementation in a model, when an external test tool is involved in test execution.

The scope of implementing model based testing is not limited to any particular platform or operating system. More work should be done to implement MBT in new SW platforms. The implementation method followed in this thesis through case studies could be a good example of implementing MBT in new platforms or operating systems in coming days. One such new area of test automation would be Windows Phone in near future. Nokia has already announced on endorsing Windows Phone as a primary OS in their upcoming phones. It means there would be a need of significant level of testing once a new product gets finalized. Various UI level automation practices could be adopted. Once UI design for Windows Phone gets finalized, the milestone of implementing MBT could be achieved by following three specific steps. First, we need to build a test plugin which runs on both host and phone (for example: qttas server with QT in case of Symbian OS). This test plugin will help in generating IP address in SUT which in turn gets recorded to XML file of a host. Second, we can quickly write an adapter application that will contain the definitions for SUT and logic for executing keywords etc. Third, SUTs are flashed with proper SW releases and are running with stable UI ready for being tested.

Having said a lot about model based testing, this thesis also analyzed the practical benefits and risks associated with its use. The most effective way to proceed with MBT deployment at this phase could be introducing this testing paradigm through small pilots and providing education and training about the tools to the target people.

REFERENCES

[1] Quality quotes collection. [www] Accessed: 26th December 2010. Web: http://www.worldofquotes.com/topic/Quality/1/index.html.

[2] Dale Emery and Elizabeth Hendrickson. Quoted from an article in *Software Testing.* Published: July 2, 2007. [www] Accessed: 2nd January 2011. Web: http://software-testing-zone.blogspot.com/2007/07/software-testing definition.html.

[3] Jan Tretmans. Conference presentation in "*Introduction to Model Based Testing"*. Slide page 8. [www] Accessed: 27th January 2011. Web: http://www.cs.ru.nl/~tretmans/.

[4] Software QA and Testing Resource Center. Article in *Software QA interview questions and answers.* [www] Accessed: 3rd January 2011. Web: http://sqa.fyicenter.com/FAQ/Software-QA-Testing/.

[5] ApTest (software testing specialists). Article in "*Types of Software Testing"*. [www] Accessed: 1st February 2011. Web: http://www.aptest.com/testtypes.html.

[6] Mark Utting and Bruno Legeard. A book titled *Practical Model based Testing- A Tools Approach*. Morgan Kaufmann, 2007.

[7] RedStone Software Inc. Article in "*Black-box vs. White-box Testing: Choosing the Right Approach to Deliver Quality Applications*". [www] Accessed: 1st January 2011.
Web: http://www.testplant.com/download_files/BB_vs_WB_Testing.pdf.

[8] Rob Davis PE. Article on "*Grey Box Testing*". Accessed: 1st January 2011. Web: http://www.robdavispe.com/free2/software-qa-testing-test-tester-2210.html.

[9] DACS Gold. *Model Based Testing*. Published: October, 2010 Vol. 13 No.3. Accessed: 2nd February 2011.

[10] Software Testing Forum, Article in "*Manual and Automation testing Challenges*", Accessed: 2nd January 2011. Web: http://www.softwaretestinghelp.com/manual-and-automation-testing-challenges/.

[11] Ruby Community. An *open source programming language*. [www] Accessed: 5th February 2011. Web: http://www.ruby-lang.org/en/.

[12] Bruno Legeard and Mark Utting. *"Model Based Testing-Next Generation Functional Software Testing".* Vol.12, No.4, Published: January 2010. Accessed: 5th February 2011.

[13] Utting, M. (2005) Position paper: Model based testing. In: Verified Software: Theories, Tools, Experiments, October 10–13, Zurich. Accessed: 5th May 2011. Web: http://www.cs.waikato.ac.nz/~marku/papers/utting_mbt_position.pdf.

[14] Schulz, S., Honkola, J. & Huima, A. (2007). *Towards model based testing with architecture models.* In: 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, March 26–29, Tucson, AZ, U.S.A.

[15] Olli-Pekka Puolitaival. Article in *Model based Testing tools*. VTT, Technical Research Centre of Finland. [www] Accessed: 25th February 2011. Web: http://www.cs.tut.fi/tapahtumat/testaus08/Olli-Pekka.pdf.

[16] Utting, M., Pretschner, A. & Legeard, B. (2006). A taxonomy of model based testing. Working Paper Series ISSN 1170-487X. Working Paper: 04/2006.

[17] Olli-Pekka Puolitaival. *Adapting model based testing to agile context*. VTT Publications 694. Master´s Thesis, Oulu University, 2008.

[18] Rational Software Corporation. Article in "Role: Test Designer". [www] Accessed: 9th January 2011.
Web: http://rup.hops-fp6.org/process/workers/wk_tstds.htm.

[19] Origsoft, original software. Article *in "Relieving the Software Testing Bottleneck"*. [www] Accessed: 10th January 2011.
Web: http://www.origsoft.com/solutions/relieving-testing-bottlenecks/.

[20] Henri Heiskanen. *"Debug Support for model based GUI testing".* Master's thesis, Tampere University of Technology, 2009.

[21] Wikipedia. An article in *Smartphone.* [www] Accessed: 10th January 2010. Web: http://en.wikipedia.org/wiki/Smartphone

[22] Forum.Nokia portal. *Introduction to TDriver.* [www] Accessed: 22nd February 2011. Web: http://projects.forum.nokia.com/Testabilitydriver.

[23] TEMA development team in Tampere University of Technology, Department of Software Systems. *Introduction to TEMA Toolset.* [www] Accessed: 25th January 2011. Web: http://tema.cs.tut.fi/intro.html.

[24] Antti Jääskeläinen. "*Design, implementation and use of a test model library for GUI testing of smartphone applications,"* Doctoral dissertation, Tampere University of Technology, Tampere, Finland, Jan. 2011, number 948 in publications.

[25] Mikko Satama, Event Capturing Tool for Model based GUI Test Automation. Master's thesis, Tampere University of Technology, September 2006.

[26] Forum.Nokia portal. Introduction to *Visualizer Code Editor*. [www] Accessed: 23rd February 2011. Web: http://projects.forum.nokia.com/Testabilitydriver.

[27] Forum.Nokia portal. Wiki page for Tdriver. [www] Accessed: 1st March 2011. Web: https://projects.forum.Nokia.com/Testabilitydriver/wiki/TDriverHelp.

APPENDIX A: SAMPLE LOG FILE

When model execution by test engine stops, ends or gets completed, the log files are generated and can be downloaded for further debugging purpose. Log files records the individual information regarding model execution. This section of appendix shows the sample of such log file generated while test execution was in run.

```
Starting testing: Thu Apr 28 08:54:12 2011
Creating test configuration ... Done
Composing model ... Done
Engine parameters
--model=parallellstsmodel:combined-rules.ext
--guidance=randomguidance
--guidance-args=randomseed:406190073
--coveragereq=
--Adapter-args=port:9092
--Adapter=socketserverAdapter
--testdata=file:sut_qt.td,file:sut_qt2.td,
--actionpp=localspp
--actionpp-args=file:sut_qt.td:N8.csv,lang:sut_qt.td:en
--verify-states=1
.END engine parameters
0428085413.532 Logger: FDLogger prepared for run 2011-04-28-08-54-13
0428085413.532 InitEngine: Initializing
0428085413.537 ParallelLstsModel: Model component 0 loaded from
'sut_qt2/rm/camera%20-%20Camera_cases-rm.lsts.nolayout'
0428085413.538 ParallelLstsModel: Model component 1 loaded from
'sut_qt2/rm/TaskSwitcherGEN-rm.lsts.nolayout'
0428085413.539 ParallelLstsModel: Model component 2 loaded from
'sut_qt2/camera%20-%20Camera_cases-awgt.lsts'
0428085413.539 ParallelLstsModel: Model component 3 loaded from
'sut_qt2/TaskSwitcherGEN-awgt.lsts'
0428085413.540 ParallelLstsModel: Model component 4 loaded from
'sut_qt/rm/camera%20-%20Camera_cases-rm.lsts.nolayout'
0428085413.541 ParallelLstsModel: Model component 5 loaded from
'sut_qt/rm/TaskSwitcherGEN-rm.lsts.nolayout'
0428085413.542 ParallelLstsModel: Model component 6 loaded from
'sut_qt/camera%20-%20Camera_cases-awgt.lsts'
0428085413.543 ParallelLstsModel: Model component 7 loaded from
'sut_qt/TaskSwitcherGEN-awgt.lsts'
0428085413.543 ParallelLstsModel: Model component 8 loaded from
'TargetSwitcher-awgt.lsts'
0428085413.544 ParallelLstsModel: Model component 9 loaded from
'TargetSwitcher-rm.lsts'
0428085413.545 ParallelLstsModel: Model component 10 loaded from
'Synchronizer-awgt.lsts'
0428085413.545 ParallelLstsModel: Model component 11 loaded from
'Synchronizer-rm.lsts.nolayout'
0428085413.552 DummyCoverage: Initialized
```

```
0428085413.553 ParallelLstsModel: Action words: sut_qt2/camera%20-
%20Camera_cases:end_awTakePicture sut_qt2/camera%20-
%20Camera_cases:end_awChangeMode
TargetSwitcher:end_awActivate<sut_qt2> sut_qt/camera%20-
%20Camera_cases:end_awBackFromPicture Synchronizer:end_awVerifysut_qt
TargetSwitcher:end_awActivate<sut_qt> Synchronizer:end_awVerifysut_qt2
sut_qt2/camera%20-%20Camera_cases:end_awLaunchCamera sut_qt/camera%20-
%20Camera_cases:end_awLaunchCamera sut_qt/camera%20-
%20Camera_cases:end_awTakePicture sut_qt/camera%20-
%20Camera_cases:end_awChangeMode sut_qt2/camera%20-
%20Camera_cases:end_awExitChangeMode sut_qt/camera%20-
%20Camera_cases:end_awExitChangeMode sut_qt2/camera%20-
%20Camera_cases:end_awBackFromPicture sut_qt2/camera%20-
%20Camera_cases:end_awCloseCamera sut_qt/camera%20-
%20Camera_cases:end_awCloseCamera
0428085413.553 TestData: Initialized, initial symbols: first, any,
next
0428085413.554 TestData: Loaded from 'sut_qt.td' symbols sut_qt
0428085413.555 TestData: Loaded from 'sut_qt2.td' symbols sut_qt2
0428085413.555 TestData: Ready to run with 5 symbols.
0428085413.555 Guidance: Using parameters {'randomseed': 406190073}
0428085413.582 LocalizationPP: Reading data from file 'N8.csv'.
0428085413.582 LocalizationPP: Found languages: en
0428085413.582 LocalizationPP: The language of device 'sut_qt' changed
to 'en'
0428085413.582 LocalizationPP: 0 data rows read.
0428085413.582 LocalizationPP: Localization index has now 0 values.
0428085413.583 LocalizationPP: The language of device 'sut_qt' changed
to 'en'
0428085413.583 Adapter: Using parameters {'bindaddr': '', 'maxlen':
5000, 'port': 9092, 'timeout': None}
0428085413.583 Adapter: Initializing socket.
0428085413.583 Adapter: Waiting for a connection from a client.
0428085436.898 Adapter: A client ('127.0.0.1', 44595) connected.
0428085436.898 InitEngine: Starting initialization, going through 0
model(s).
0428085436.898 InitEngine: Initialization done.
0428085436.899 TestEngine: Local time zone UTC+3.0
0428085436.899 TestEngine: Testing starts from state (0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0)
0428085436.901 TestEngine: Step     :     1 Covered:  0.0000 % Next:
WAKEtgtsCANWAKE<sut_qt2>
0428085436.901 TestEngine: Executing: WAKEtgtsCANWAKE<sut_qt2>
0428085436.901 TestEngine: New state: (0, 0, 0, 2, 0, 0, 0, 0, 1, 0,
0, 0)
0428085436.903 TestEngine: Step     :     2 Covered:

                        ..............
                        ..............
                        ..............
                        ..............
```

The list of events will keep on growing until the test execution ends.

……The End……